



Symbian OS

Communications Programming



Second Edition

The Symbian logo, featuring the word "symbian" in a lowercase, sans-serif font. The letters "i" and "a" are stylized with a blue dot and a blue horizontal line, respectively.

Iain Campbell



Symbian OS Communications Programming

2nd Edition

By

Iain Campbell

With

**Dale Self, Emlyn Howell, Ian Bunning, Ibrahim Rahman, Lucy
Caffery, Malcolm Box, Matthew Elliott, Natasha Ho, Pierre
Cochart, Tim Howes, Twm Davies**

Reviewed by

**Chris Notton, Dan Handley, David Harper, David Singleton,
Donald Page, Graeme Duncan, Ian Bunning, John Roe, Malcolm
Box, Tim Howes**

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



John Wiley & Sons, Ltd

Symbian OS Communications Programming

2nd Edition

Symbian OS Communications Programming

2nd Edition

By

Iain Campbell

With

Dale Self, Emlyn Howell, Ian Bunning, Ibrahim Rahman, Lucy Caffery, Malcolm Box, Matthew Elliott, Natasha Ho, Pierre Cochart, Tim Howes, Twm Davies

Reviewed by

Chris Notton, Dan Handley, David Harper, David Singleton, Donald Page, Graeme Duncan, Ian Bunning, John Roe, Malcolm Box, Tim Howes

Head of Symbian Press

Freddie Gjertsen

Managing Editor

Satu McNabb



John Wiley & Sons, Ltd

Copyright © 2007 Symbian Software Ltd

Published by John Wiley & Sons, Ltd The Atrium, Southern Gate, Chichester,
West Sussex PO19 8SQ, England

Telephone (+44) 1243 779777

Email (for orders and customer service enquiries): cs-books@wiley.co.uk
Visit our Home Page on www.wileyeurope.com or www.wiley.com

All Rights Reserved. No part of this publication may be reproduced, stored in a retrieval system or transmitted in any form or by any means, electronic, mechanical, photocopying, recording, scanning or otherwise, except under the terms of the Copyright, Designs and Patents Act 1988 or under the terms of a licence issued by the Copyright Licensing Agency Ltd, 90 Tottenham Court Road, London W1T 4LP, UK, without the permission in writing of the Publisher. Requests to the Publisher should be addressed to the Permissions Department, John Wiley & Sons Ltd, The Atrium, Southern Gate, Chichester, West Sussex PO19 8SQ, England, or emailed to permreq@wiley.co.uk, or faxed to (+44) 1243 770620.

Designations used by companies to distinguish their products are often claimed as trademarks. All brand names and product names used in this book are trade names, service marks, trademarks or registered trademarks of their respective owners. The Publisher is not associated with any product or vendor mentioned in this book.

This publication is designed to provide accurate and authoritative information in regard to the subject matter covered. It is sold on the understanding that the Publisher is not engaged in rendering professional services. If professional advice or other expert assistance is required, the services of a competent professional should be sought.

The Bluetooth™ word mark and logos are registered trademarks owned by Bluetooth SIG, Inc. and any use of such marks by Symbian Software Ltd is under license. Other trademarks and trade names are those of their respective owners.

Other Wiley Editorial Offices

John Wiley & Sons Inc., 111 River Street, Hoboken, NJ 07030, USA

Jossey-Bass, 989 Market Street, San Francisco, CA 94103-1741, USA

Wiley-VCH Verlag GmbH, Boschstr. 12, D-69469 Weinheim, Germany

John Wiley & Sons Australia Ltd, 42 McDougall Street, Milton, Queensland 4064, Australia

John Wiley & Sons (Asia) Pte Ltd, 2 Clementi Loop #02-01, Jin Xing Distripark, Singapore 129809

John Wiley & Sons Canada Ltd, 6045 Freemont Blvd, Mississauga, Ontario, L5R 4J3, Canada

Wiley also publishes its books in a variety of electronic formats. Some content that appears in print may not be available in electronic books.

Library of Congress Cataloging-in-Publication Data

Campbell, Iain.

Symbian OS communications programming / Iain Campbell, with Dale Self . . .
[et al.]. – 2nd Edition.

p. cm.

Previously published: Symbian OS communications programming / Michael J.
Jipping, 2002.

Includes bibliographical references and index.

ISBN 978-0-470-51228-9 (pbk. : alk. paper)

1. Symbian OS (Computer file) 2. Operating systems (Computers) 3. Data
transmission systems. I. Jipping, Michael J. Symbian OS communications
programming. II. Title.

QA76.76.063J56 2997

055.4'482 – dc22

2007011028

British Library Cataloguing in Publication Data

A catalogue record for this book is available from the British Library

ISBN: 978-0-470-51228-9

Typeset in 10/12pt Optima by Laserwords Private Limited, Chennai, India

Printed and bound in Great Britain by Bell & Bain, Glasgow

This book is printed on acid-free paper responsibly manufactured from sustainable
forestry in which at least two trees are planted for each one used for paper production.

Contents

Contributors	ix
About the Authors	xi
Acknowledgments	xv
Symbian Press Acknowledgements	xvii
Section I: Introduction and Overview	
1 Introduction	3
1.1 What is in this Book	3
1.2 What isn't in this Book	4
1.3 Expected Level of Knowledge	6
1.4 Structure of this Book	7
1.5 To which Versions of Symbian OS does the Information in this Book Apply?	8
1.6 Example Applications	9
1.7 Reading Guide	9
1.8 Other Sources of Information	9
1.9 The History of Symbian OS Communications	10
1.10 Summary	12
2 Overview	13
2.1 Low-level Functionality	14
2.2 High-level Functionality	19
2.3 Summary	23

Section II: Low-level Technology and Frameworks

3	An Introduction to ESOCK	27
3.1	Overview of ESOCK	27
3.2	Into Practice	52
3.3	Summary	62
4	Bluetooth	63
4.1	Bluetooth Technology Overview	63
4.2	Bluetooth in Symbian OS	82
4.3	Example Symbian OS Bluetooth Application	112
4.4	AV Protocols and Profiles	118
4.5	Summary	124
5	Infrared	125
5.1	Introduction	125
5.2	Infrared Overview	125
5.3	IrDA in Symbian OS	129
5.4	Summary	153
6	IP and Related Technologies	155
6.1	IP Networks Overview	156
6.2	IP Networks and Symbian OS	160
6.3	Network Bearer Technologies in Symbian OS	163
6.4	Using the Network Connection	175
6.5	Information Gathering and Connection Management	193
6.6	Quality of Service	197
6.7	Summary	203
7	Telephony in Symbian OS	205
7.1	Overview	206
7.2	Using the ETel ISV API	207
7.3	Restrictions and Considerations	211
7.4	Summary	214

Section III: High-level Technology and Frameworks

8	Receiving Messages	217
8.1	Example Application – Summary Screen	218
8.2	The Message Server	220
8.3	The Message Store	223
8.4	Messaging Application Design and Implementation	233

8.5	Receiving Application-specific SMS Messages	235
8.6	Summary	240
9	Sending Messages	241
9.1	Examples Provided in this Chapter	242
9.2	SendAs Overview	242
9.3	Services/Accounts	246
9.4	Technical Description	246
9.5	Using the UI Platform Send Dialogs	250
9.6	A Brief Background to MTMs	255
9.7	The Flickr MTM	257
9.8	The Flickr Data MTM	260
9.9	The Flickr UI MTM	262
9.10	Flickr Client MTM	263
9.11	The Flickr Server MTM	265
9.12	MTM DLLs and Platsec	268
9.13	FlickrMTM Shared Settings	269
9.14	Installation of an MTM	269
9.15	Summary	271
10	OBEX	273
10.1	OBEX Overview	273
10.2	OBEX in Symbian OS	289
10.3	Summary	341
11	HTTP	343
11.1	HTTP Overview	343
11.2	Getting Started: Creating a Session	344
11.3	Creating and Submitting a Transaction	347
11.4	Supplying Body Data	349
11.5	Monitoring a Transaction	350
11.6	Cancelling a Transaction	353
11.7	Closing a Transaction	353
11.8	Stringpool	353
11.9	Proxy Support	355
11.10	Cookie Handling	356
11.11	HTTP Connection Configuration	356
11.12	Platform Security	361
11.13	Filters	361
11.14	Summary	364
12	OMA Device Management	365
12.1	Introduction	365
12.2	Device Management In Symbian OS	366
12.3	OMA Device Management Essentials	367

12.4	The Example DM Adapter	372
12.5	Summary	387

Section IV: Development Tips

13	Setting Up for Development	391
13.1	Bluetooth	391
13.2	IrDA	394
13.3	Network Connections for IP	396
13.4	Telephony	403
13.5	'Help, help, my serial port's been stolen'	404
13.6	Summary	406
14	The Future	407
14.1	Better Networks	407
14.2	Better Interaction	409
14.3	Better Services	410
14.4	The End	411
	Appendix A: Web Resources	413
	Appendix B: Authorizing FlickrMTM to Use Your Flickr Account	415
	Appendix C: SendWorkBench.app Guide	419
	Index	421

Contributors

Head of Symbian Press

Freddie Gjertsen

Authors

Iain Campbell
Dale Self
Emlyn Howell
Ian Bunning
Ibrahim Rahman
Lucy Caffery
Malcolm Box
Matthew Elliott
Natasha Ho
Pierre Cochart
Tim Howes
Twm Davies

Symbian Press Editorial

Managing Editor
Satu McNabb

Reviewers and additional contributors

Chris Notton
Dan Handley
David Harper
David Singleton
Donald Page
Graeme Duncan
Ian Bunning
John Roe
Malcolm Box
Tim Howes

About the Authors

Iain Campbell, lead author

Iain joined the comms team (as it then was) in Symbian in 2001, working on Symbian OS v6.1, v7.0 and v7.0s for the Nokia 7650, Sony Ericsson P800 and Nokia 6600, respectively. After spending a year working in the Bluetooth team creating the Symbian OS PAN profile implementation, he moved to Symbian's Technical Consulting group where he has spent the last three years helping Symbian's licensees and partners build Symbian OS-based phones. As part of this Iain has been involved in many aspects of Symbian OS – from debugging components at all levels of the system, through advising on adaptation to particular hardware platforms, to high-level system design. Iain received an MEng in Information Systems Engineering from Imperial College, London, and enjoys spending his spare time finding out how things work.

Malcolm Box

Malcolm first joined Psion Software in 1998, shortly before it became Symbian. His first job was writing the kernel for the Ericsson R380 phone, following which he led the design and implementation of the Symbian OS Bluetooth stack. Subsequently he's worked in the System Architecture group, Symbian's reference design team and with licensees as a senior consultant. He has previously co-authored *Symbian C++ for Mobile Phones* and contributes to various open-source projects. He would like to thank his wife, Judith, and children Franz and Abigail for their support and patience during the writing of this book.

Ian Bunning

Ian attended Trinity Hall at the University of Cambridge, where he gained an MA in Computer Science. On graduating in 2001 he joined the Shortlink team at Symbian, and soon became the expert on the IrDA subsystem. Since then he has also worked on a number of OBEX projects, as well as a smaller number of Bluetooth projects – the main one being part of the initial implementation of Bluetooth PAN profile. He is currently focusing on USB, but frequently supports maintenance work on IrDA and OBEX. Out of work hours, Ian is a keen photographer, and also makes items of jewellery.

Lucy Caffery

Lucy has been at Symbian since 2000, where she has worked for the Licensee Product Development team helping UIQ licensees to create Symbian products. Starting out as a Bluetooth specialist, she became Head of the Comms Porting group in LPD, a team which specializes in consultancy in all areas of the Symbian OS Comms subsystem. More recently Lucy has become the Deputy Head of LPD. Lucy has been involved in comms on all the UIQ devices that have shipped to date: Sony Ericsson P8xx, P9xx, M600i, W950i and P990, Motorola A92x, A1000 and M1000.

Pierre Cochart

Pierre Cochart graduated from King's College London in 2000. He then joined Symbian as a graduate in the telephony team to help with the development of the 7.0 OS release. In 2003 he joined Licensee Product Development group to work in the Comms Porting group where he assisted customers with software development in various areas of comms. Pierre is now responsible for handling the communications area for the Japanese licensees.

Twm Davies

Twm joined Symbian as a graduate in 1999. Twm has had a varied career within Symbian, initially working as a developer of the 'crystal' messaging application which provided the UI to the Nokia communicator range, then as a technical consultant for Motorola, Nokia and significantly the technical lead on the first non-Nokia S60 handset, the Siemens SX1. Twm currently works as Product Manager for performance. Twm graduated

from Cardiff University with a First Honours Computer Science BSc. Interests outside of work include collecting mispronunciations of his name, scuba diving, Vespas and he runs a web site selling his art works.

Matt Elliott

Matt joined Symbian in 2004 as a software engineer, and has spent his time at Symbian in the Device Provisioning team. He graduated with a BEng in Digital Electronics from the University of Kent, and coming from a hardware background still misses his soldering iron (but not the burnt fingers). Matt would like to thank all the past and present members of the Device Provisioning team for their carefully worded criticism/help, and his long suffering girlfriend Elaine.

Natasha Ho

Natasha joined Symbian in 1998, where she worked on the development of the Ericsson R380. Since then, she has contributed to almost every UIQ smartphone including the Motorola A920 and A1000, the Arima U300 and more recently the Sony Ericsson P800, P900, M600i, W950i and P990i. She has worked on various parts of the Symbian OS but now likes to concentrate solely on networking. Prior to Symbian, Natasha worked at Motorola designing and writing software for the GSM and GPRS cellular infrastructures. Natasha graduated from University College London with a BSc in Computer Science.

Emlyn Howell

Emlyn Howell has worked on various technologies within Symbian over the past seven years including messaging and telephony. He is currently the Comms Architect for the Reference Designs team. He lives and works in Cambridge.

Tim Howes

After studying for a PhD in the effects of indirect lightning strikes on power lines, Tim joined Symbian Software, where for seven years he has worked primarily within the Bluetooth area. Within the Bluetooth SIG, Tim represents Symbian on the Bluetooth Architecture Review Board, and contributes to the Core Specification, Audio Video and Medical Devices Working groups. Despite the high technology area Tim works in, he has a strong interest in mechanical timepieces.

Ibrahim Rahman

Ibrahim has been at Symbian for eight years. Working as a software developer in areas including email and HTTP.

Dale Self

Dale started work for Psion Software in mid-1998, which transformed to Symbian about a week later. Initially working in the messaging team on an IMAP4 mail client, he later moved to the PAN team where he has worked with Bluetooth, OBEX and USB technologies ever since. During this time he has seen a great deal of growth; both in Symbian, and, sadly, in his waist measurement.

Acknowledgements

Firstly we'd like to thank the Symbian Press team who helped put this book together, especially Satu, who kept us working on it until it was finished – without her it would probably be sitting half-written on various hard disks around Symbian.

Secondly we'd like to thank all of our wives, husbands, partners and significant others for putting up with us whilst we hid away in the evenings and weekends writing the material for this book.

I'd like to thank Apple for creating a computer that's a pleasure to use – it made the whole editing process so much less painful.

And finally I'd like to extend an additional thank you to my wife Chris, who put up with me not moving from in front of the computer for a month whilst I pulled the book into shape.

Iain Campbell

Symbian Press Acknowledgements

Symbian Press would like to thank Iain for his patience during this project and for the countless hours he spent polishing the text into perfection.

We'd also like to thank the authors Dale, Emlyn, Ian, Ibrahim, Lucy, Malcolm, Matt, Natasha, Pierre, Tim and Twm for their dedication and hard work, and all the reviewers for their time and willingness to share their technical knowledge.

Section I

Introduction and Overview

1

Introduction

Welcome to the updated edition of *Symbian OS Communications Programming*! In this book we'll introduce you to much of the major communications functionality in Symbian OS and demonstrate how to perform common tasks in each area.

For this new edition we've started from scratch to produce chapters that are relevant to you as developers. Each chapter gives background information on the technology where necessary, an overview of the functionality provided in Symbian OS, and descriptions or examples of how to use the Symbian OS APIs. In cases where APIs or implementation differ between Symbian OS-based devices this is noted, and when the user interface platforms work differently then we'll show you what those differences are, or at least point you in the direction of some documentation that does.

1.1 What is in this Book

In this book we focus on using and extending Symbian OS functionality using the native C++ APIs. Whilst it is also possible to use Java to develop applications for Symbian OS devices, we do not cover that in this book. We also focus on APIs available in standard UIQ3 and S60 3rd edition SDKs – thus engineers at Symbian's licensees and partners will want to look elsewhere for details on the internals of the Symbian OS subsystems that we describe. However, the material in this book is suited to all developers – at third parties, Symbian's licensees and Symbian's partners – who wish to use the functionality described.

This book should also prove useful to newcomers to Symbian OS in the device creation community, providing a high-level overview of the communications side of Symbian OS, and an idea of how it all fits together. However, this is not likely to be sufficient for creating a device, for that you will need to look elsewhere.

There are three main user interfaces supported on Symbian OS – MOAP, S60 and UIQ. At present, only S60 and UIQ allow developers to extend functionality natively in the aftermarket, so we concentrate on those platforms in this book.

S60 and UIQ have, in some places, differences in the way they choose to use and expose certain Symbian OS functionality. As a result, some details given in this book differ between the different UI platforms. Where this is the case we will highlight this, along with tips on how to use the functionality on each platform. In some cases functionality might have an alternate implementation on a given platform, in which case we will point you to the appropriate developer documentation. In other cases, it might be missing entirely, which might mean you need to reconsider your development plans. In cases where functionality is missing or not yet exposed, it is possible that it will be available in a later release of that UI platform – check with the appropriate developer website for more information in these cases.

The scope of this book is quite broad – not only will we discuss the core communications functionality in Symbian OS – Bluetooth, IrDA, TCP/IP and telephony, but we also look at some of the main areas where those technologies are employed – the messaging framework and plug-ins, the HTTP stack, the Object EXchange (OBEX) stack and the OMA Device Management system. Therefore whether you need access to communications functionality at a high or a low level, there should be something in this book for you.

1.2 What isn't in this Book

Symbian's licensees have a lot of flexibility when creating a device – as is necessary in a market where there is plenty of differentiation between products. As such, the supported feature set in any given device depends greatly on the market segment at which that device is aimed. You can expect to find that some features are not supported in given devices – either where they are not suitable, or cost-effective enough to be included.

Equally, some devices have leading-edge features that have been developed for differentiation – in these cases, the generic implementation developed when the feature becomes widely available may differ from the original one, which is normally highly tailored for the lead device. Throughout this book we describe the generic implementations – the ones you can rely upon to remain compatible beyond the initial implementation. Therefore it is best to use these implementations wherever possible to minimize, or eliminate, the amount of rework your application requires when deploying it to a new device.

However, you may wish to use some of those leading-edge features, perhaps because your application is targeted specifically at the device containing them. In that case, we advise you to go directly to the device manufacturer's developer website for information on using such features.

1.2.1 Technologies not Covered in this Book

There are some notable absences from this book in terms of technologies that Symbian OS supports. The reasons for their omission vary, but below we'll summarize the main technologies that are missing.

USB

Symbian OS has included support for acting as a USB device (or client) since Symbian OS v7.0, and this support was backported to v6.1 for certain devices. However, extending this support is difficult due to the nature of the USB controllers used in devices. Typically these have a limited number of USB endpoints. By the time most devices are shipped, their USB configuration is such that most, if not all, of those endpoints are in use. This means that third-party extensions would not be possible, as they would not have any resources available with which to implement their functionality. As a result, Symbian defines the APIs required to extend the USB implementation as partner-only.

The one case where USB functionality might be of use to developers is the USB virtual serial port (more accurately, the USB ACM class). However, current devices are rarely configured to leave an available virtual serial port – in most cases they are already in use, providing the interface to use the device as a modem, or for use with debug agents. Check the documentation for the devices you are targeting to see if there is a USB serial port available – if there is, then it can be accessed via the RComm interface.¹

Real-time protocol and real-time control protocol (RTP and RTCP)

The Symbian OS implementation of RTP has been supported since Symbian OS v8.1, and has been shipping in devices since v9.1. However, it is not until v9.2 that it is present in both S60 and UIQ devices – in v9.1 it is currently only present in devices using UIQ3.

Various problems led to us leaving RTP and RTCP out of this book. Firstly, the RTCP API is currently marked as partner-only. Secondly, the Symbian OS RTP implementation currently lacks a few key features. There

¹ The Symbian OS Library contains more information on use of the RComm interface and opening a USB virtual serial port.

is no support for authentication or encryption of RTP packets – in fact, there is currently no support for any profiles, including the Secure Real-time Transport Protocol (SRTP) – this is left to applications to implement. Even if an application wished to try and implement SRTP, low-level cryptography functionality in Symbian OS is currently exposed via partner-only APIs. As a result, it is very difficult to deploy RTP in situations where secure audio support is required and the underlying transport doesn't provide its own security. In practice, this would mean that you would need to use IPSec for RTP over UDP and Bluetooth link layer security for any Bluetooth links using RTP, which may not be compatible with any system you are trying to integrate with.

Hopefully all of these problems will be addressed, and future versions of this book can include information on RTP. In the meantime, for those of you that wish to experiment, documentation can be found in the UIQ version of the Symbian OS Library, under 'Mm Protocols RTP'.

Session initiation protocol (SIP)

Symbian OS v9.2 contains a SIP stack. A similar implementation has been available as part of the S60 UI platform for a number of releases now, and has appeared in some, but not all, S60 devices. However, as with RTP, we have chosen not to write about SIP until the situation regarding availability in devices and recommended APIs has been resolved. For more information on SIP in v9.2, consult the S60 3rd edition, Feature Pack 1 documentation.

1.3 Expected Level of Knowledge

This book assumes you have had some experience of developing for Symbian OS, and that you are aware of the basic paradigms – active objects, the client–server framework, and descriptors and have an understanding of the basic types of Symbian OS classes (e.g., C-, R-, T-, M-classes) and how they behave.

If you are new to Symbian OS, or want to take a refresher course in these concepts, there are several books that you could read, including *Symbian OS C++ for Mobile Phones*, and *Symbian OS Explained*, which offers an excellent guide to the basics of programming for Symbian OS.

In addition, we assume you're familiar with the concepts of the platform security model used in Symbian OS since v9.0. We describe capabilities required to perform operations in this book, as well as the other impacts that platform security has, but we don't dedicate space to the underlying concepts. For more details, you can read the *Symbian OS Platform Security* book, although it goes into a lot more depth than is required to understand our discussions in this book. There is a sample

chapter from the book available on the Symbian website – this contains a good description of the three key concepts of platform security, and is well worth reading. There is also a good FAQ about platform security available from the Forum Nokia website, <http://www.forum.nokia.com>.

In terms of required communications knowledge, we assume you're familiar with the usage of SMS, MMS, email and HTTP, so have an understanding of the features they provide. More mobile-specific technologies, such as OBEX, Bluetooth, IrDA and Device Management, have more background information as we assume that these technologies may not be as familiar to most developers, especially those from a PC background.

We also assume that you are able to set up a general development environment for Symbian OS development – you have installed the appropriate UI platform software development kit (SDK), an integrated development environment (IDE), if you choose to use one, and are at the stage where you can build software for both emulator (again, if you choose to use it – we strongly advise that you do) and the target platform (whichever phone(s) you choose to test on). However, we don't require you to know about setting up the emulator for comms development – we have a chapter towards the end of the book explaining how to do this for various different technologies. Obviously testing on actual hardware is also extremely important, and in some cases the only way to test your application. In this case though we have little to say in terms of setting it up – the hardware features of your phone are fixed, and most of the configuration issues we discuss in relation to the emulator simply don't apply – the settings have already been configured for your particular device.

1.4 Structure of this Book

This book is divided into four main sections. The first section – chapters 1 and 2 – aim to give an overview of the book and its contents. It also provides an introduction to the communications functionality in Symbian OS, and a high-level overview of how it all fits together.

The second section – chapters 3 through 7 – covers lower level communications technologies. Roughly speaking, these are technologies that exist, in Symbian OS, below a socket interface – Bluetooth, TCP/IP, IrDA – or technologies that provide simple data links, such as virtual serial ports,² or other core services such as telephony. Each of these technologies acts as the underlying layer for the functionality covered in Section III.

² Real serial ports, of the RS-232 variety, have died out completely from phones, and these days only exist on development boards. As this book is targeted at developers using phones rather than development boards as their target hardware, we only discuss serial ports in the context of the virtual serial ports provided by Bluetooth and IrDA.

The third section – Chapters 8 through 12 – covers higher level technologies, such as the messaging framework and plug-ins (for SMS, MMS and email), the SendAs service, OBEX, hypertext transfer protocol (HTTP) and OMA Device Management (OMA DM). As these technologies build upon those discussed in Section II, some APIs from that section will be revisited here – for example, the HTTP APIs expose the `RConnection` API. So whilst each chapter tries to stand alone, it might be necessary to read an earlier chapter to understand all that is being described. See section 1.7 for a more comprehensive reading guide.

The fourth section – Chapter 13 – contains practical information on developing with Symbian OS. This includes information on setting up the development environment for various types of comms-related development. Much development can be done using the Symbian OS emulator – we discuss various ways of connecting the emulator to an IP network, and using an IR pod, Bluetooth hardware and a phone with the emulator. Finally, there is a brief chapter about future developments, covering new features or areas of technology that are likely to be interesting in the future. If enough of these changes take place, it will be time for us to start writing a new book!

The sample code provided in this book is designed to be dropped straight into your application – this means it doesn't use the shortcuts seen in other examples, where the code will work in the standalone example but needs rework before you can use it in your application. Where there is additional work needed, for example where errors need to be handled in an application-specific way or we have omitted some code for the sake of clarity, this will be shown clearly – this means you can spend more time working on the interesting part of your application, and less figuring out why our code doesn't work in your application!

1.5 To which Versions of Symbian OS does the Information in this Book Apply?

This book targets Symbian OS v9.1 and v9.2. Symbian OS v9.1 is the version used in S60 3rd edition and UIQ 3.0 phones. In version 9.0 of Symbian OS, a new security model was introduced which affected the usage of many APIs. This book describes the APIs after the security changes were made. Some APIs have changed very little, some have had methods replaced, and some have been removed completely.

In cases where the API that we are discussing is significantly different in an older version of Symbian OS, information will be presented in a box like this one. Reference will be made to the old API, allowing you to look up the information in the appropriate version of the Symbian OS Library.

Information referring to future versions of Symbian OS – including that which applies to v9.2 but not v9.1 – will be presented in a similar style.

1.6 Example Applications

Most chapters have example applications to demonstrate the functionality we describe. You can download the source code from the Symbian website: <http://developer.symbian.com/main/academy/press>. Whilst the example apps have all been developed on S60 3rd edition or UIQ3 emulators and devices, the user interfaces provided by some apps use a simple text shell. This is in order to focus on developing the communications side of the application rather than designing a nice UI.

1.7 Reading Guide

Whilst we've tried to make each chapter stand alone, inevitably some chapters have dependencies on others in this book. These are our suggestions for reading each chapter.

You are reading Chapter 1 now, so that's covered. We suggest that everyone takes a look at Chapter 2 to understand the basics of the system structure.

Chapter 3, covering ESOCK, is rather abstract – we suggest you read one of Chapters 4, 5 or 6 (Bluetooth, IrDA, TCP/IP) first, then go back and look at chapter 3 to understand the basic framework.

Chapter 7, Telephony, stands alone.

Chapters 8 and 9 fit together well, as they cover the messaging functionality from two different angles. The example set of MTMs in chapter 9 use the example code in chapter 11 for some functionality.

Chapter 10, OBEX, requires some understanding of the material covered in chapters 4 and 5 – this is clearly indicated in the text.

Chapter 11, HTTP, refers to the `RConnection` API, which is explained in chapter 6.

The example in Chapter 12, OMA Device Management, manipulates settings for the example app in Chapter 6, so it's worth having a look at the application to see which settings need to be managed remotely.

1.8 Other Sources of Information

Several times in this book we refer to the Symbian OS Library (formerly the Symbian OS Developer Library). This is available online at

<http://developer.symbian.com/main/oslibrary>, and is also integrated into the documentation that comes with S60 and UIQ SDKs.

Information on specific UI platforms, as well as additional example code for Symbian OS, is available from the appropriate developer website www.forum.nokia.com or <http://developer.uiq.com>.

1.9 The History of Symbian OS Communications

To finish up this chapter we'll give you some background on the development of the communications functionality in Symbian OS, and take a quick glance at how it has evolved over time – starting from the initial implementation by Psion.

The initial history of Symbian OS communications is closely tied to that of the Psion Series 5 organizer – one of the original palmtop computers. In 1997, Psion PLC released the original Series 5 organizer – a revolutionary 32-bit, ARM-based PDA. This was a follow on from their 16-bit Series 3 organizer, itself a popular product in UK and some parts of Europe in the early 1990s. A major strength of the Series 5 was an in-built suite of PIM applications. The other was an increasingly strong suite of communications-related functionality added during the product's lifetime. The initial device contained an infrared port to beam information between devices over IrDA (but not using IrOBEX, which wasn't standardized in time for the initial Series 5) as well as a serial port for connecting to PCs and modems. To access this functionality two of the components still in use today were first introduced – c32 (more recently referred to as the serial server) and ESOCK, which provides the Symbian OS socket API.

Released after the original organizer as an installable software component, the Message Suite added a messaging subsystem with POP3, SMTP and fax capabilities to the device.

In 1999, the follow up to the original Series 5 was released – the Series 5mx. This increased the amount of RAM and ROM in the device, providing space for more functionality. OBEX was added to replace the original Psion-proprietary IR transfer protocol – providing standards-based object transfer to other (non-Psion) devices. Also included was a new messaging architecture – the same as the one as you see today – designed to allow extension of the messaging functionality of the device using aftermarket plug-ins called Message Type Modules (MTMs), the first of which was an IMAP4 implementation.

Later came a web browser from Opera Software AB. By this time, the Psion organizer was sporting as much communications functionality as some PCs, all in a much smaller format and running off two AA batteries.

In 1998, Symbian Ltd was formed – a joint venture between Ericsson, Motorola, Nokia and Psion. The staff of Psion Software Ltd transferred

to Symbian Ltd, and started working to produce the first Symbian-based mobile phone – the Ericsson R380. This was based off a Unicode version of the ER5 release that powered the Series 5mx, and marked the start of the ‘one-box’ era in Symbian – the first time that the communications solution did not require a separate phone or modem in addition to the Symbian OS-based device, which had traditionally been referred to as the ‘two-box’ model.

One notable feature lacking from the R380 was the ability to install software. In 2001, the first open Symbian OS-based device was released by Nokia to replace their 9100-series communicator – the Nokia 9210. Again, this was a one-box design, with integrated GSM functionality.

By this point, Symbian was working on adding both GPRS and Bluetooth functionality, which eventually shipped in Symbian OS v6.1. The first product to take advantage of this was the Nokia 7650 – most notable for being the first Symbian OS-based phone with an integrated camera. This feature meant it also included support for multimedia messaging (MMS) as part of the messaging functionality. As part of the introduction of Bluetooth, OBEX was extended to run over the new transport, and had OBEX authentication added. Symbian OS v6.1 also saw the release of the new IP-based connectivity solution for Symbian OS – replacing the old Psion Link Protocol (PLP) for connecting the device to PCs. As well as serving as the basis for a series of Nokia phones, this version of the OS also went into a series of sophisticated handsets released by Fujitsu for the DoCoMo network in Japan.

The first Symbian OS v7.0 device was the Sony-Ericsson P800, released in 2002. New functionality in this release of Symbian OS included an HTTP stack, as well as the first release of the hybrid IPv4/IPv6 stack. The ETel multimode APIs were introduced, as was the first version of SyncML.

Symbian OS v7.0s was first used in the Nokia 6600. Building on v7.0, the new release added advanced multihoming capabilities to the IP stack – allowing multiple simultaneous IP-based network connections with no risk of IP address range clashes or ambiguous routing situations. This functionality is still a step ahead of many operating systems today, many of which have problems when attached to multiple IP networks. This feature is especially important if there are any ambiguities between the networks, such as address ranges that are operational in both networks – as is often the case where private IPv4 addresses are in use. Symbian OS v7.0s also introduced support for OBEX over USB, and the ability for OBEX to receive objects to file rather than RAM.

Symbian OS v8.0 saw major enhancements to the Bluetooth APIs for link management, often referred to as the Bluetooth v2 architecture. OBEX gained APIs for user-defined headers and the ability to use double buffering when transferring objects.

Symbian OS v8.1 saw the introduction of Bluetooth PAN profile, supporting the GN and U roles – although it was never shipped in a

device until v9.1 when it appeared in UIQ3 devices. Other changes included the SSL/TLS library being rewritten, providing new functionality such as client authentication, and including new cipher suite support including AES-based cipher suites. This version went into a series of phones released by Fujitsu and Mitsubishi in the Japanese market, as well as the Nokia N70, N72 and N90.

Symbian OS v9.0 never shipped publicly – instead acting as an interim release to allow Symbian OS licensees to adopt the platform security model.

Symbian OS v9.1 has been an extremely popular version of Symbian OS – shipping in over 12 S60 and three UIQ device models so far. It saw the introduction of the multithreaded version of ESOCK, capable of running each protocol module in its own thread to allow increased independence in scheduling of protocols. Also introduced was support for the Bluetooth A2DP profile – whilst the profile itself is not implemented by Symbian OS, much of the core technology is contained within the Symbian OS Bluetooth implementation. A new API on ESOCK, `RSubConnection`, was also introduced – initially to allow quality of service (QoS) functionality to be exposed.

Features added in Symbian OS v9.2 are covered in the rest of this book.

1.10 Summary

In this chapter we've summarized what information is contained in this book and explained how it is structured. We've also mentioned which topics and technologies are not in this book, and where to go to find information on them. You should have some idea of the level at which this book is pitched – we assume you have some basic Symbian OS development experience, and are in a position to be able to create applications, but don't expect you to know everything about communications technologies or their implementation in Symbian OS.

The next chapter introduces the major communications subsystems in Symbian OS, and shows how they fit together. If you're interested in a specific technology, and are happy about basic communications concepts and the structure of communications in Symbian OS, feel free to skip ahead to the appropriate chapter. Otherwise, read on for an introduction to Symbian OS communications functionality.

2

Overview

In this chapter we'll present a brief overview of the different parts of Symbian OS we're going to talk about in this book. Detailed explanations of how things work will be left to the individual technology chapters, but here we'll take a quick look at what each area does.

First we have something of an aside, but on an important topic – that of binary compatibility (BC).

Symbian OS v9.0 introduced a major BC break with versions 8.1 and earlier, due to the introduction of a new (standardized) application binary interface (ABI). Therefore applications compiled against v8.1 and earlier of Symbian OS will not run on v9.1 and later. In addition to this, the platform security model requires some changes to applications using communications functionality. Depending on the application this might be as simple as adding a CAPABILITY statement to the MMP file.

To start with we take a quick look at a very high-level model of how a Symbian OS-based phone is constructed in order to see where various components fit in. This isn't to say every phone looks like this, but it is a good enough model to help understand the different layers that exist within Symbian OS.

The signalling stack (also referred to as the 'baseband' or 'cellular modem') is provided by the manufacturer of the device, so Symbian OS contains a series of adaptation layers to allow standard Symbian OS APIs to be adapted to the specific signalling stack in use. Today, all shipping products use either a 2.5G GSM/GPRS or 3G UMTS (aka W-CDMA, aka 3GSM) signalling stack. In Figure 2.1, you can see the layers that form the adaptation coloured grey – we'll talk a little about each of these throughout this chapter.

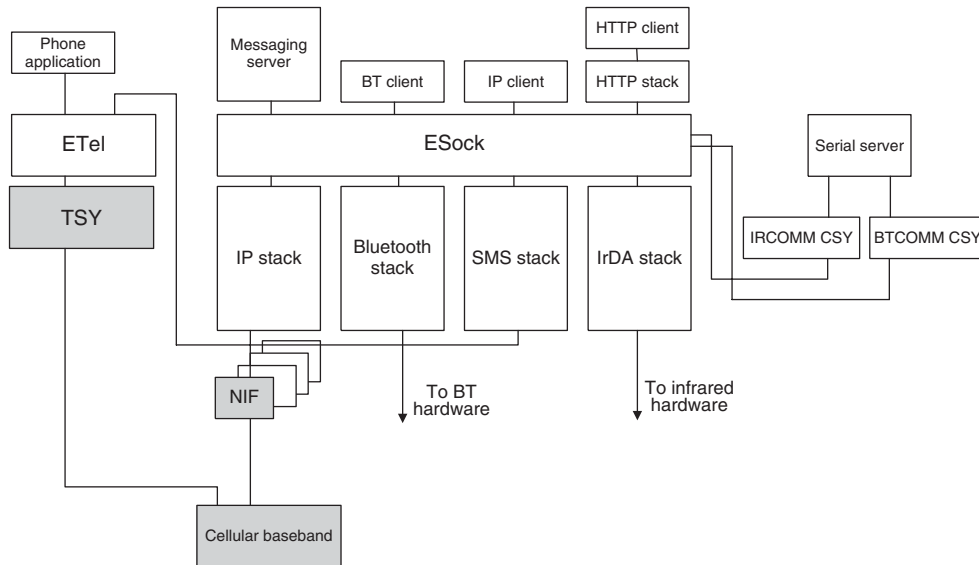


Figure 2.1 High-level overview of Symbian OS communications architecture

2.1 Low-level Functionality

We'll split our overview into two main sections, following those that the whole book uses, in order to partition the system more clearly. In this first part of the chapter we'll talk about the technologies in section 2 – the lower level areas such as the Bluetooth, IrDA and IP stacks, along with telephony. In the second part we'll discuss more about messaging, OBEX, HTTP and OMA Device Management.

There are a couple of key frameworks involved in Symbian OS communications, so let's look at those first. Not everything fits into these frameworks (most of the higher level functionality in section 3 does not, for example) but they provide the foundation for the topics discussed in section 2.

Symbian OS undergoes continual development, so details on the internals are subject to change. Public interfaces (publishedAll in Symbian parlance), which make up most of the content of this book, are subject to strict compatibility controls; however, internals are likely to evolve over time. When we're discussing internal details, which we are going to do in this chapter, we'll make it clear so you know that the information may change.

2.1.1 The Root Server and Comms Provider Modules

Root server and comms provider modules are internal to Symbian OS¹ – therefore this whole section should be considered informational.

Many of the lower level communication technologies in Symbian OS run inside the c32exe (often abbreviated to ‘c32’) process. In the initial versions of Symbian OS (or EPOC32, as it was back then) the serial server² was the initial thread in this process, and provided functionality to start further threads within the c32 process. Today, the root server performs this function, loading and binding comms provider modules (CPMs) into the c32 process. The serial server, ESOCK server and ETel server are all now loaded as CPMs by the root server. All this is done under the control of the c32start process, which is responsible for loading and configuring the CPMs based on the contents of various CMI files found within c32start’s private directory.

The exact mechanisms used to load ESOCK, ETel and the serial server aren’t normally of any interest to us. However, as we’ll see in Chapter 13, there are some circumstances when using the emulator when we might want to alter which CPMs get loaded, so it’s useful to keep the above information in mind when setting up development environments for communications programming.

2.1.2 ESOCK

ESOCK provides both the framework for, and the interface to, various lower level protocols in Symbian OS. For example, core parts of the Bluetooth stack, most of IrDA, most of the TCP/IP stack, and much of the SMS stack³ exist within the ESOCK framework, and are accessed through ESOCK APIs. Figure 2.2 gives an overview of the ESOCK framework. Chapter 3 contains a lot of useful information about the public ESOCK APIs, with additional information in the technical chapters that refer to technologies implemented as ESOCK plug-ins, such as Bluetooth. As you can guess from the name, one of the key public interfaces that ESOCK provides is based upon a sockets API.

Here we’ll talk a little about what happens behind the scenes with ESOCK and the various plug-ins.

¹ Actually, the APIs are available to Symbian partners, but for the purposes of this book, that’s the same as internal to Symbian.

² ‘Serial server’ is also quite a recent term – originally it was called the c32 server – a reflection of the fact that the hardware available for communications in the original Psion Series 5 consisted of a serial port and an IrDA port, and therefore the serial server was the interface to the core communications functionality in the device.

³ Although there are a variety of places in the system where you might choose to access SMS messages – see Chapter 8 for details.

Again, this is information about internals, so is subject to change.

ESOCK plug-ins exist in DLLs with the suffix .prt, and are loaded into the ESOCK threads as part of the ESOCK startup mechanism. Specifically, ESOCK looks for ESK files in the ESOCK subfolder of the c32 private directory. These contain details of how to bind various protocol layers together, along with some protocol-specific configuration information. As with CMI files, we'll see some more details of this in Chapter 13 when it comes to setting up development environments.

Bluetooth

The Bluetooth standards include a rich and varied set of functionality, ranging from using wireless audio headsets for making telephone calls, through emulating an Ethernet network, to connecting keyboards to a host. Much of the functionality available is described in Bluetooth profiles – documents describing an end-to-end use case for a user, such as using an audio headset with a mobile phone as a hands-free kit. Those use cases tend to be implemented completely by the time a phone reaches the market. However, there are also APIs available to application programmers to implement their own Bluetooth-based applications and protocols.

Chapter 4 describes two main areas – what is required to extend a profile that Symbian implements (for example, interfacing to the remote control functionality of the Audio/Visual Remote Control Profile (AVRCP) profile to allow your application to receive commands from a Bluetooth stereo headset) and what is required to implement Bluetooth communications between devices running your application, using one of two possible protocols – L2CAP or RFCOMM.

There are also some Bluetooth profiles which can be used without any Bluetooth-specific APIs. The prime example of this is Bluetooth PAN profile, which implements a virtual Ethernet network between Bluetooth devices. In this case, the standard APIs used for creating IP connections are used, although there are some special behaviors implemented to cope

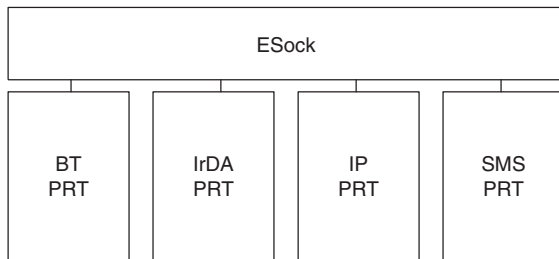


Figure 2.2 ESOCK and protocol plug-ins

with the different models used by Bluetooth and normal, wired, Ethernet networks.

Bluetooth functionality is implemented in several modules, which are split across various processes, as shown in Figure 2.3 – here we cover the most obvious ones from a developer viewpoint. The core Bluetooth stack, implemented as an ESOCK plug-in, provides the L2CAP, RFCOMM, AVDTP and AVCTP protocols. The local service discovery database is implemented as a standalone component, which sits above the Bluetooth stack. The Bluetooth manager is a Symbian OS server that provides access to the local device's Bluetooth registry. And finally the RFCOMM CSY is the plug-in to the serial server that allows access to RFCOMM through the RComm interface.

One related component is the remote control server, RemCon, which implements a generic system for distributing commands from remote devices, typically some form of remote control, to local applications. It's mentioned here as it is the server for the AVRCP implementation, which receives commands from the remote control buttons present on some Bluetooth stereo headsets. These are then routed through a UI platform-specific target selector plug-in (TSP) to the appropriate application for processing. There are more details on this system in Chapter 4.

There are some components that we haven't mentioned, most notably the host controller interface (HCI) which provides the interface from the Bluetooth stack to the hardware, but these are adaptation specific and typically do not present APIs for general use.

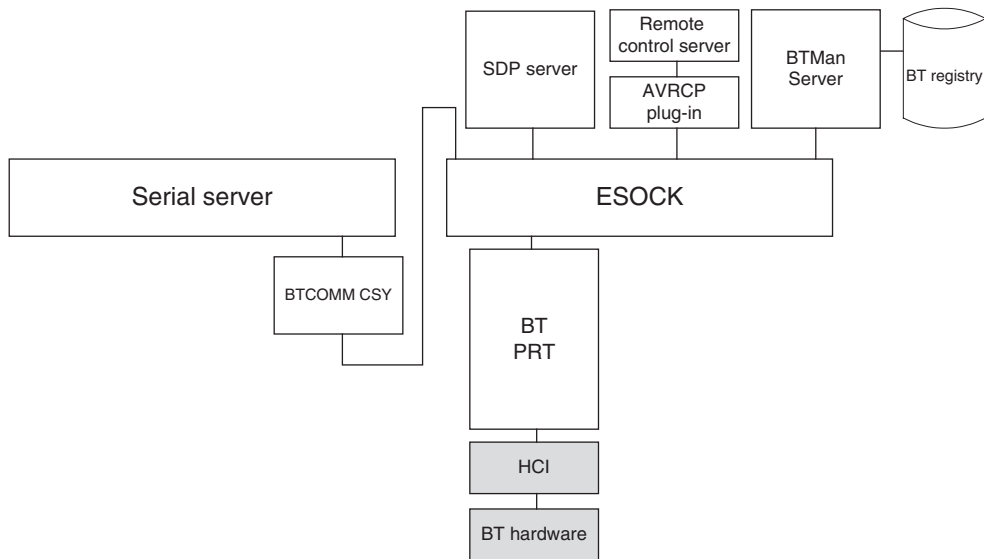


Figure 2.3 High-level bluetooth architecture

IrDA

IrDA is an older technology than Bluetooth, providing line-of-sight communications over a range of around one metre. The specifications themselves define fewer profile details than Bluetooth – the major profile being OBEX, which is covered in a separate chapter. As such, the Symbian OS support is primarily centred on the core specifications. When using IrDA, the two main protocols of interest are IrCOMM, for serial port emulation, and TinyTP, for stream-style communication. IrDA also provides device discovery and simple service discovery capabilities. Chapter 5 describes the implementation of the IrDA protocols in more detail.

While the IrDA committee has defined extensions to the core IrDA specifications, Symbian OS only provides the basic services. Key omissions are support for Fast Infrared, Fast Connect and IrSimple.

IP and network interfaces

Some of the details of the internals of the networking subsystem, for example network interfaces for particular network technologies, and the IP hooking mechanism, are internal (actually partner-only again) and therefore subject to change.

Symbian OS contains a hybrid IPv4/IPv6 stack, implemented as an ESOCK plug-in. It implements all the protocols you'd expect – TCP, UDP, IPv4 and IPv6, and delegates the domain name system (DNS) functionality to a separate module – the Domain Name Daemon, which performs DNS lookups and caching for all applications on the platform. Despite this (internal) delegation, the interface to the DNS service is provided through the standard ESOCK APIs.

The IP stack has the ability to load plug-ins that can intercept and modify packets as they pass through the stack; however these interfaces are partner-only, and so are not described in this book.

Access to particular network technologies is performed using a combination of a network interface (NIF) and an agent (AGT). The normal separation of responsibility is that the network interface is a lightweight packet processor that sits on the data path, whereas the agent contains the logic to make and manage the connection. With certain network protocols though, the line is blurred, and therefore certain NIFs contain code to perform configuration of the interface – the Point-to-Point Protocol (PPP) NIF being a prime example.

On the control side, there is also the network interface manager (NIFMAN) and the network controller (NETCON), which, together with ESOCK, provide the framework for creating and managing IP connections.

NIFMAN also provides a system for plugging in network configuration mechanisms independently of underlying network technologies – at present the DHCP client implementation is the only example of such a plug-in.

SMS

The short message service (SMS) stack is also a plug-in to the ESOCK framework. It provides various lower level services involved in sending an SMS, including fragmenting messages that are too long to fit in a single protocol data unit (PDU). It also provides APIs to allow applications to receive SMS messages before they are delivered to the messaging store – a useful feature for pushing information to your application without triggering an irrelevant indication to the user. These APIs are covered further in Chapter 8.

2.1.3 ETel

ETel provides the lowest level interface to the telephony functionality within Symbian OS. Below ETel is the realm of the device-specific adaptation, which converts ETel API calls into whatever form is necessary for the specific baseband stack that is in use – this is the job of the TSY (Telephony SYstem module).

Because ETel provides a very low-level API, Symbian implement a wrapper over this called ‘ETel 3rd party’, which provides a limited subset of the functionality available in the ETel API in an easy-to-use form. A discussion of this API and the functionality available is in Chapter 7.

2.1.4 Serial Server

The serial server provides, perhaps unsurprisingly, the interface to serial ports, both real and virtual. A series of plug-ins, called Comms SYstem plug-ins (CSYs, see Figure 2.4), provide access to specific technologies – RS-232 physical serial ports (rarely found on phones today), and Bluetooth, infrared and USB virtual serial ports. Some products contain additional, product-specific CSYs for communicating with other parts of the system, such as the cellular modem, but these are rarely useful to developers. Since USB is not covered in this book, and RS-232 ports are so rarely encountered, virtual serial ports over Bluetooth and infrared are covered in the respective technology chapters.

2.2 High-level Functionality

Now it’s time to look at the functionality built on top of the low-level services that we’ve just discussed.

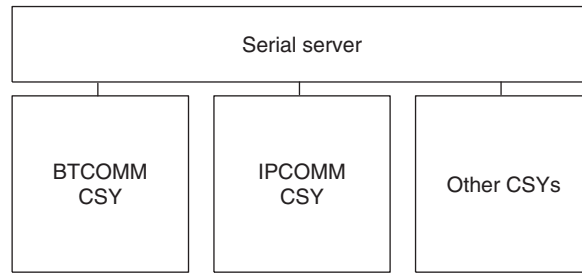


Figure 2.4 Serial server and CSYs

2.2.1 Messaging

Messaging functionality in Symbian OS is based around the messaging server, and a series of plug-ins called Message Type Modules (MTMs), as shown in Figure 2.5. By default, Symbian OS-based phones ship with a rich set of MTMs built-in, including ones for SMS, MMS, IMAP4, POP3, SMTP and OBEX over Bluetooth and IrDA (for transferring objects between devices at short range). This set can be extended – there are MTMs available from many push email providers, as well as from other third parties.

A set of plug-ins to support a single messaging technology consists of four different MTMs – a server MTM, a client MTM, a UI MTM and a UI data MTM. Together they form a complete solution for adding a new message transport to the messaging architecture; although complex implementations may choose to provide some of the functionality of the UI MTM in separate applications. The purpose and responsibilities of each type of MTM are described in the second part of Chapter 9.

On top of messaging is built the SendAs service, which allows applications to send data to remote devices using an abstract API – this allows the set of technologies that can be used to send messages to be extended,

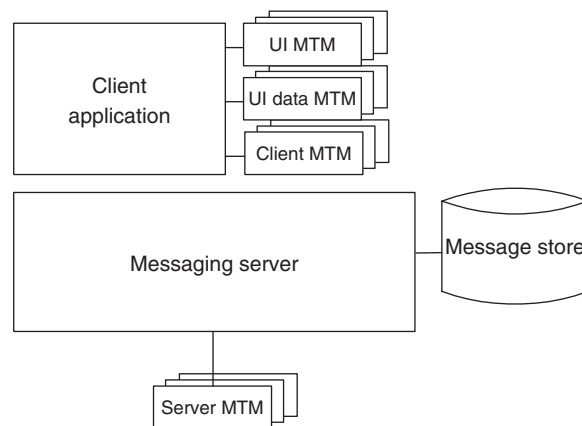


Figure 2.5 Message server and MTMs

based on MTMs available on the system that support the SendAs service, without the application using the service having to change.

Chapter 8 covers receiving messages, and contains details of using various messaging APIs to access the message store. Chapter 9 covers sending messages using the SendAs service, including the UI-specific wrappers around it, as well as describing how to create a new set of MTMs to enable additional message transports to be added to the SendAs service.

2.2.2 OBEX

OBEX was originally created by the IrDA specifications body as an efficient means of exchanging objects, and has since been extended to support a variety of other transport protocols, such as Bluetooth and USB. OBEX performs a similar role to the more popular HTTP protocol. There are many similarities between the two – both are request-response protocols, both are designed to transfer arbitrary data between devices, and both use the concept of headers describing the content and bodies containing it.

OBEX is designed to be easier to implement in resource-constrained devices than HTTP. It is also designed to be more efficient in terms of bandwidth, using a binary encoding scheme for headers rather than a text-based one. And the OBEX protocol itself is stateful, unlike HTTP which is fundamentally stateless and uses the additional concept of cookies to maintain state.

In Symbian OS, the OBEX stack is loaded into the client's process. As well as the core stack, there are a number of transport plug-ins as seen in Figure 2.6 – for IrDA, Bluetooth and USB. In order to use the OBEX stack the client's process must have the appropriate capabilities to perform the operations that OBEX attempts – the required capability is `LocalServices` for existing transports.

OBEX is used to transport data in a number of use cases – most commonly in the use case it was originally designed to fulfil – the transfer of objects such as vCards between devices. However, it is also used in a number of other Bluetooth specifications, including advanced image transfer and printing, and also as a transport protocol for OMA SyncML.

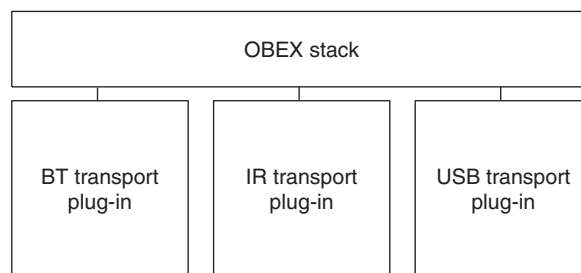


Figure 2.6 The OBEX stack and transport plug-ins

Chapter 10 describes the Symbian OS implementation of OBEX, showing how to connect transport sessions, as well as build, parse and exchange objects.

2.2.3 HTTP

HTTP is most familiar as the protocol that serves up web pages on the Internet. Symbian OS contains an HTTP stack that is available for any application to use. The HTTP stack is loaded in the client's process, and can be extended with client-supplied plug-ins. As it runs in the client's process the client must have the appropriate capabilities to perform the operations that the HTTP stack attempts – currently this is the capability `NetworkServices`. Figure 2.7 shows the high level architecture of the HTTP stack.

As mentioned in the OBEX section, at a high-level HTTP and OBEX fulfil similar roles – they both provide a system for transferring arbitrary objects between a client and a server.

The Symbian OS HTTP stack supports HTTP/1.0 and HTTP/1.1, and as part of HTTP/1.1 it supports persistent connections and pipelining. It also supports basic and digest authentication, and HTTP over TLS using the 'https' URI scheme. A number of HTTP error codes, such as those for redirection are handled transparently. Chunked transfer encoding is supported, however, content encodings are not, by default, implemented, although that's not to say there isn't a filter providing them on a given UI platform.

The HTTP stack can be extended at runtime by the use of plug-ins called *filters*. These filters sit in the path between the client and the TCP socket and can listen for various events and modify ongoing transactions. The Symbian OS standard filter set includes a redirection filter, an authentication filter and a validation filter. UI platforms tend to introduce additional filters, including cookie handling and caching, some of which may get added to the default set of filters that are loaded when a client loads the HTTP stack. The HTTP stack is covered in more detail in Chapter 11.

2.2.4 OMA Device Management

With the increasing number of services available on mobile phones, one of the key technologies in making them easier to use is a method for remote management of configuration settings. For both network operators and medium and large companies, the ability to remotely provision and update settings on the user's device has the potential to greatly reduce the cost of deploying new solutions. This is the purpose of OMA Device Management.

Symbian OS contains a framework to which Device Management adapters (DM adapters) can be added to manage settings for specific applications using a generic device management protocol. This

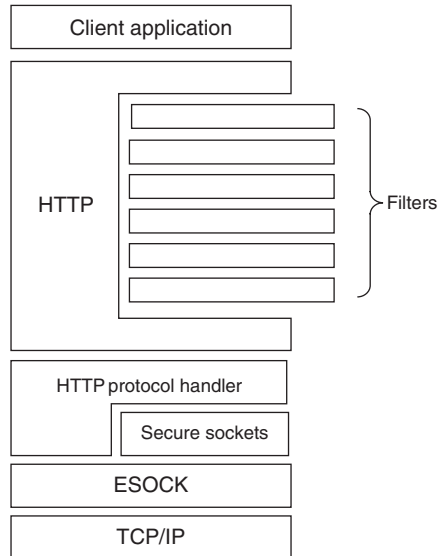


Figure 2.7 The HTTP stack and filter plug-ins

means application authors, by providing a DM adapter that exposes the configuration settings within their application in a standard form, can enable it for remote management.

The device management framework relies on the underlying SyncML framework to provide services common to device management and data synchronization. The common SyncML framework then provides a series of transport protocols over which it can be run. DM adapters plug into the DM framework, and provide standardized access to a variety of underlying settings stores – each settings store potentially being unique to a particular application.

2.3 Summary

That ends our overview of the different technologies described in this book. Hopefully you now have a high-level understanding of what each area does and how the different parts fit together. Next we'll start looking at each area in-depth. If there's a particular area you're interested in, feel free to skip straight to that chapter, although it might be worth consulting the reading guide in Chapter 1 to see if there is any suggested pre-reading for a given chapter.

Section II

Low-level Technology and Frameworks

3

An Introduction to ESOCK

This chapter introduces one of the core communications frameworks of Symbian OS, ESOCK. We will show you how to use the ESOCK APIs to connect to a service, send and receive data, and how to allow remote hosts to connect to your own program.

The example code presented in this chapter uses the ESOCK APIs, using the TCP protocol as an example – full details on how these APIs are used with a specific protocol (e.g., Bluetooth, IP or IrDA) are found in the relevant technology chapter.

In this chapter we will be creating a set of wrapper classes for the raw sockets API both to demonstrate how sockets are used on Symbian OS and as a basis for your own programs.

3.1 Overview of ESOCK

ESOCK provides two main services: the sockets API, including host resolvers and network-based databases, for making connections with remote systems and transferring data; and the connection management API for selecting, configuring and monitoring connections.

3.1.1 The Sockets API Overview

The sockets API, found in the `es_sock.h` header file, provides the core functionality for communicating over a variety of network protocols, including TCP/IP, Bluetooth and IrDA. This includes:

- name and address services to discover available communications partners, or to map names to network addresses
- making connections to remote services

- receiving connections from remote hosts
- sending and receiving data.

By providing a common idiom for accessing diverse network protocols, ESOCK makes it easier for programmers to adapt their code to new communication technologies as they come along. Porting still isn't completely transparent, thanks to the specific details of each technology, but it's much easier than if every protocol had a completely different API and programming model.

History

The original sockets API was introduced in 4.2BSD Unix in 1983 and spread to other Unix variants before being adopted in other operating systems. The sockets idiom has proven enduringly popular for accessing communication services and is now used on the majority of operating systems and thus by almost all programs that need to communicate over a network.

ESOCK and the sockets API were first introduced into Symbian OS in EPOC Release 1 and have been a central part of the operating system ever since. Since v7.0s, the basic socket API has been extended to cope with the unique networking environment that phones face:

- the possibility of multiple access points to networks, which may have different costs, availability and quality of service
- certain IP-based services that can only be accessed via certain access points.

3.1.2 The Connection API Overview

A mobile phone today has a host of communication options. These include short-range technologies such as infrared (IrDA) and Bluetooth, wireless LAN (WiFi), packet-switched wide-area links such as GPRS, EDGE and WCDMA, and finally circuit-switched 'dial-up' connections.

Several of these links can be used to access an IP network, usually the Internet, but often corporate networks too, each with different performance (bandwidth, latency) and costs.

The result is that the choice of which network access method to use is more complicated than on a PC. To allow programs to manage this complexity, the ESOCK connection API allows the enumeration, configuration and selection of the various IP connections, including the ability to prompt the user to choose.

Despite being originally designed for control of connections supporting IP, the connection APIs are now being extended to support the needs of other technologies, although the set is limited as of Symbian OS v9.2.

3.1.3 Accessing ESOCK

As ESOCK is a server running in its own thread in the C32 process, any program that wants to use the ESOCK APIs must first connect to the server to provide a client/server session.

This is handled by the `RSocketServ` class, which provides the following main methods:

```
TInt Connect(TUint aMessageSlots=KESockDefaultMessageSlots);
TInt Connect(const TSessionPref& aPref,
             TUint aMessageSlots=KESockDefaultMessageSlots);
TVersion Version() const;
TInt NumProtocols(TUint& aCount);
TInt GetProtocolInfo(TUint anIndex, TProtocolDesc& aProtocol);
TInt FindProtocol(const TProtocolName& aName, TProtocolDesc& aProtocol);
void StartProtocol(TUint anAddrFamily, TUint aSockType,
                  TUint aProtocol, TRequestStatus& aStatus);
void StopProtocol(TUint anAddrFamily, TUint aSockType,
                  TUint aProtocol, TRequestStatus& aStatus);
```

The `RSocketServ::Connect()` methods provide the connection to ESOCK that all other ESOCK APIs rely on. It's usually fine to use the default for the `aMessageSlots` parameter. If you find in testing that your code is getting `KErrServerBusy` errors then you can increase the number of message slots.

Since Symbian OS v9.1, ESOCK uses multiple threads to run different protocols and because a Symbian OS client-server connection must be made to a specific thread, by default the main ESOCK thread forwards messages to the right thread for the protocol in use. If you know in advance which protocol you'll be using, the `TSessionPref` parameter can provide a speed-up by allowing a direct connection to the right thread, saving a thread switch. This is only a hint to ESOCK, so you can still use any protocol with this session if needed. Since thread switching is very fast, you're highly unlikely to notice any difference, which means the most usual form used for the connection is:

```
RSocketServ ss;
TInt err;
err = ss.Connect(); // Default slots & protocol
// check error and handle
```

One useful facility is the `StartProtocol()` call to asynchronously load a protocol. Programs can use this to avoid calls to `RSocket::Open()` blocking while the protocol starts up, which can be a lengthy process. Typically though, most protocols that you require are already loaded, so using this call often isn't necessary.

Despite some versions of the Symbian OS documentation claiming this needs the `NetworkControl` capability, there is no capability required to make this call.

The other `RSocketServ` methods allow the enumeration, finding and stopping of protocols. As a program needs to understand the details of how to use a protocol before it makes use of it, these are rarely used.

3.1.4 RSocket and Friends

The Symbian OS sockets API is provided by the `RSocket` class, found in `<es_sock.h>`. The methods on this class break down into various sections:

1. Creating a new socket (and setting it up for use).
2. Connecting to a service.
3. Receiving connections from a peer.
4. Sending data to a peer.
5. Receiving data from a peer.
6. Control functions: options and ioctls.
7. Miscellaneous: naming, transferring and information.

We'll look at each of these areas in turn below, but first it's important to understand the different types of sockets and protocols understood by Symbian OS.

Types of sockets

While the sockets API provides a consistent *syntax* for using various protocols, the *semantics* of the APIs are partially dependent on the specific protocol. Different protocols provide different services, guarantees and options, and this is reflected in the behavior of the sockets APIs.

One major difference is in how data is sent and received on a socket. ESOCK provides four different socket types:

1. Stream – A stream socket provides a byte-oriented, reliable data channel. 'Byte-oriented' means data can be written or read in arbitrary lengths; 'reliable' means that data is guaranteed to arrive in order at the receiver without duplication or loss, or not to be received at all. Examples: TCP (TCP/IP), RFCOMM (Bluetooth).
2. Sequenced packet – A sequential packet socket provides a reliable, packet-based interface. 'Packet-based' means data must be written in blocks of length no larger than the maximum packet size, and is read as individual packets. This means that the sender must know the maximum size of the outgoing packets and respect this limit. On the receive side, each call to `Recv()` returns the data from a single packet, so if the buffer provided is smaller than the size of the packet's data the remaining data will be thrown away, and the next read will return the next packet's data. This behavior can be overridden using the `KSockReadContinuation` flag in the `Recv()` call, in which

case any remaining data will be returned in the next `Recv()` call. Examples: L2CAP (Bluetooth), Tiny TP (IrDA).

3. Datagram – A datagram socket provides a packet-based interface that may be reliable or unreliable, depending on the protocol. In the unreliable case, the sender can usually make no assumption about whether the data arrives at all, out of order or duplicated at the receiver. The send/receive interface works as for sequential packet sockets. Examples: UDP (TCP/IP), IrMUX (IrDA).
4. Raw – A raw socket provides low-level access to the packets being received/transmitted and the ability to freely manipulate these.

Types of protocols

Communications protocols are divided into two families: connection-oriented and connectionless.

Connection-oriented protocols (and their associated sockets) require that the socket is connected to the peer before data is sent or received, whereas connectionless sockets allow data to be sent without a connection having been established first.

Connectionless protocols generally (but not always) provide unreliable datagram interfaces, as it is difficult to provide reliable data transfer without a connection, for example to know when to clean up memory used to store data until it is acknowledged by the receiver.

Socket addresses

One of the key classes involved in the use of `RSocket` is the `TSockAddr`. This is a standard base class for all socket addresses supported by Symbian OS. By default it only has one attribute – a port – but each technology creates its own classes derived from it to add methods for storing the appropriate type of address, e.g., `TBTSockAddr`, `TInquirySockAddr`, `TInetAddr`, `TIrdaAddr`.

The use of this base class is the reason that, providing you to perform a lookup using a DNS name and do not try to manipulate the underlying address, use of IPv4 and IPv6 is transparent for all applications – even applications written before IPv6 was created!

Lifecycle of a socket

Figure 3.1 shows the lifecycle of a socket.

Opening a socket Sockets are opened using the various `RSocket::Open()` calls:

```
TInt Open(RSocketServ& aServer, TInt addrFamily, TInt sockType,
          TInt protocol);
```

```

TInt Open(RSocketServ& aServer,TUint addrFamily,TUint sockType,
          TUint protocol, RConnection& aConnection);
TInt Open(RSocketServ& aServer,TUint addrFamily,TUint sockType,
          TUint protocol, RSubConnection& aSubConnection);
TInt Open(RSocketServ &aServer,const TDesC& aName);
TInt Open(RSocketServ& aServer);

```

As you can see, each of these take a `RSocketServ` parameter, which must be a connected session to ESOCK. The first four versions of `Open()` associate the socket with a specific protocol and/or connection/subconnection (described below). The protocol can be identified either by the `<addrFamily, sockType, protocol>` tuple, or by name. The fifth version allows a blank socket to be created, which is used for incoming connections.

To find the correct `<addrFamily, sockType, protocol>` values, you can either look in the Symbian OS documentation, or use the protocol enumeration methods of `RSocketServ` to see what's available on your device. Table 3.1 shows some of the common values for Symbian OS v9.x.

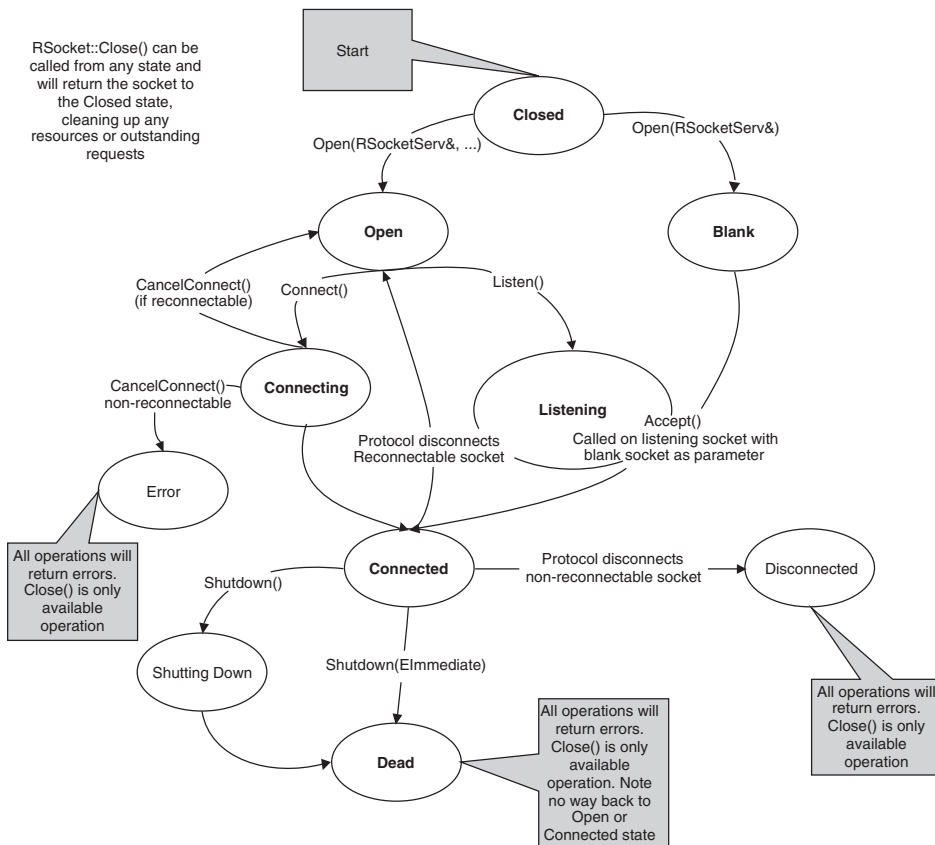


Figure 3.1 The lifecycle of a socket

Table 3.1 Protocol values

Family	Protocol	Protocol name	addrFamily	sockType	Protocol
IPv4/6	TCP	tcp	KAfInet	KSockStream	KProtocolInetTcp
IPv4/6	UDP	udp	KAfInet	KSockDatagram	KProtocolInetUdp
IPv4/6	ICMP	icmp	KAfInet	KSockDatagram	KProtocolInetIcmp
IPv6	ICMP	icmp6	KAfInet6	KSockDatagram	KProtocolInet6Icmp
Bluetooth	RFCOMM	RFCOMM	KBTAddrFamily	KSockStream	KRFCOMM
Bluetooth	L2CAP	L2CAP	KBTAddrFamily	KSockSeqPacket	KL2CAP
Bluetooth	AVCTP	AVCTP	KBTAddrFamily	KSockDatagram	KAVCTP
Bluetooth	AVDTP	ACDTP	KBTAddrFamily	KSockDatagram	KAVDTP
Bluetooth	Link Manager	BTLinkManager	KBTAddrFamily	(Host resolver)	KBTLinkManager
IrDA	Tiny TP	IrTinyTP	KIrdaAddrFamily	KSockSeqPacket	KIrTinyTP
IrDA	IrMUX	Irmux	KIrdaAddrFamily	KSockDatagram	KIrmux

Connecting a socket As all communication involves at least two parties, there are two directions in which a connection can be made between your program and a peer. Either the connection is initiated by your program (an ‘active’ connection), or your program waits to receive connections from a peer (a ‘passive’ connection). These roles have also traditionally been called ‘client’ and ‘server’, reflecting the normal practice of having a client program connecting to a server to request a service. However, once the connection has been made there may be no difference between the two ends and the link can be used in any way. For example, in a multiplayer game both ends may be identical software which sends game data back and forth with no ‘client’ or ‘server’ roles.

Active connections An active connection to a peer is usually made using the `RSocket::Connect()` call, although this doesn’t have to be used for connectionless sockets:

```
void Connect(TSockAddr& anAddr, TRequestStatus& aStatus);
void Connect(TSockAddr& anAddr, const TDesC8& aConnectDataOut,
            TDes8& aConnectDataIn, TRequestStatus& aStatus);
```

The address of the device to connect to is passed in the `TSockAddr` parameter. Since the addressing information needed by each protocol is

different, most protocols provide a class derived from `TSockAddr` that contains the specific information needed by that protocol. This can then be passed to the `RSocket::Connect()` call as a `TSockAddr`. `Connect()` can be used with both connection-oriented and connectionless protocols – with the latter it simply sets the default address for outgoing packets to be sent to, as there is no connection to initiate.

`Connect()` is an asynchronous call, reflecting the fact that creating a connection may take a substantial amount of time – potentially several seconds or more, depending on the technology in use. Your program should be prepared to deal with these delays, and the variation between different bearers.

The second version of `Connect()` allows for additional data to be sent/received during the connection. This is only supported by some protocols – you can determine if the protocol you’re using supports this by using the `RSocket::Info()` or `RSocketServ::GetProtocolInfo()` methods to retrieve details on the protocol. These are returned in a `TProtocolDesc` structure, of which the `iServiceInfo` member provides various flags:

```
// iSock is a opened socket
TProtocolDesc desc;
iSock.Info(desc);
if(desc.iServiceInfo & KSIConnectData)
{
    // Protocol supports connection data - can use Connect() with data
}
else
{
    // it doesn't
}
```

Since connection data is not supported by all protocols, most programs don’t make use of this facility, because it can be replaced by making a connection and then sending the data which works for all protocols. In general it’s worth avoiding relying on quirks and unique features of a particular protocol, as in future your program may need to support other protocols.

The `iServiceInfo` field in `TProtocolDesc` has lots of other information on the protocol. The details are shown in Table 3.2.

As normal, the returned value in the `TRequestStatus` indicates if the connection was created successfully. If the value is `KErrNone`, then the socket is now connected and can be used for data transfer. If not, then the application will need to handle the error, either by retrying the connection or providing an error to the user. If your code retries the connection, it should only do so a limited number of times – it may be impossible to make the connection for a wide variety of reasons depending on the protocol.

Table 3.2 Protocol details

Flag	Bitmask	Meaning
KSIConnectionLess	0x00000001	The protocol is connectionless
KSIReliable	0x00000002	The protocol is reliable
KSIInOrder	0x00000004	The protocol guarantees in-order delivery
KSIMessageBased	0x00000008	The protocol is message based
KSIDatagram	0x00000008	The same as message based
KSIStreamBased	0x00000010	The protocol is stream based
KSIpseudoStream	0x00000020	The protocol supports a stream-like interface but maintains datagram boundaries
KSIUrgentData	0x00000040	The protocol offers an expedited data service
KSIConnectData	0x00000080	The protocol can send user data on a connection request
KSIDisconnectData	0x00000100	The protocol can send user data on a disconnect request
KSIBroadcast	0x00000200	The protocol supports broadcast addresses
KSIMultiPoint	0x00000400	The protocol supports point to multi-point connections
KSIQOS	0x00000800	The protocol supports a quality of service metric. This isn't so useful, see the RSubConnectionAPI instead
KSIWriteOnly	0x00001000	The protocol is write only
KSIReadOnly	0x00002000	The protocol is read only
KSIGracefulClose	0x00004000	The protocol supports graceful close
KSICanReconnect	0x00008000	The same socket can be reconnected if it disconnects (for whatever reason)
KSIPeekData	0x00010000	Protocol supports peeking (looking at the data without removing it from the protocol)
KSIRequiresOwnerInfo	0x00020000	Protocol is to be informed of the identity of the client (i.e., process ID, thread ID and UID) of each SAP (i.e., socket service provider) created

If the connect is taking too long, or the user changes their mind, then an ongoing connection can be cancelled using the `RSocket::CancelConnect()` call. Note that if a connection is cancelled, the socket can be reused (and `Connect()` called again) only if the protocol supports reconnection (`KSICanReconnect`). Otherwise the socket is put into an error state and all operations will return errors. At this point all that can be done is to call `RSocket::Close()`.

Passive connections Accepting incoming connections from a peer is slightly more complex than making an active connection. It involves three API calls: `Bind()`, `Listen()` and `Accept()` and two sockets. Here's the details of the three calls:

```
TInt Bind(TSockAddr& anAddr);
TInt SetLocalPort(TInt aPort);
void Accept(RSocket& aBlankSocket, TRequestStatus& aStatus);
void Accept(RSocket& aBlankSocket, TDes8& aConnectData,
            TRequestStatus& aStatus);
TInt Listen(TUint qSize);
TInt Listen(TUint qSize, const TDesC8& aConnectData);
```

The first socket is the listening socket, which is bound to the address on which you want to listen for incoming connections using `Bind()`. Then you call `Listen()` on the bound socket to start the protocol stack listening for incoming connections. Finally `Accept()` is called on the listening socket, passing the second, blank, socket (see 'Opening a socket' for how to create a blank socket). When a connection from a peer is received, the accept call completes and the blank socket is now connected and ready for data transmission. You can then call `Accept()` again with another blank socket to receive the next incoming connection.

```
class CSocketListener : public CActive
{
public:
...
void ListenL();
virtual void RunL();
...
private:
RSocketServ iSs;
RSocket iListening;
RSocket iAccept;
}

void CSocketListener::ListenL()
{
    User::LeaveIfError(iSs.Connect());
    User::LeaveIfError(iListening.Open(ss, KAfInet, KSockStream,
                                     KProtocolInetTcp));
    // addr is setup with the right address somewhere else
    User::LeaveIfError(listening.Bind(addr));
}
```

```

User::LeaveIfError(listening.Listen(4)); // Q up to four incoming
//connections
User::LeaveIfError(accept.Open(ss)); // Create a blank socket
listening.Accept(accept, stat);
SetActive();
}

void CSocketListener::RunL()
{
    if(stat.Int() == KErrNone)
    {
        // Socket is now connected
    }
}

```

The queue size parameter to `Listen()` allows the protocol stack to queue up a number of incoming connections while it waits for the program to call `Accept()`. If the queue length is exceeded, then further connections are automatically rejected by the protocol stack. For most programs, where the expected rate at which incoming connections arrive is much lower than the rate at which the program can `Accept()` each connection, then a queue size of four is plenty. For an application like a web server where there might be many clients connecting simultaneously it may be necessary to increase this.

As an alternative to the `Bind()` call, `SetLocalPort()` can be used for setting up the address if only a port number is needed, e.g., for protocols such as TCP and UDP where a device can have multiple interfaces and the program wants to listen on all of these interfaces. See Chapter 6 for more details on TCP/IP, as well as details of how to listen only on specific interfaces.

As with active connections, if the protocol supports connection data then the `Listen()` and `Accept()` calls can be used to send and receive this data respectively. Again, this is rarely used.

Remember when you no longer want to receive incoming connections don't just stop calling `Accept()`, but close the listening socket using `RSocket::Close()`. This frees the listening address for use by other programs.

Sending data Once a socket has been opened and connected (for connection-oriented protocols), it can be used for sending and receiving data. To send data, there are several calls:

```

void Send(const TDesC8& aDesc, TUint someFlags, TRequestStatus& aStatus);
void Send(const TDesC8& aDesc, TUint someFlags, TRequestStatus& aStatus,
          TSockXfrLength& aLen);
void SendTo(const TDesC8& aDesc, TSockAddr& anAddr, TUint flags,
            TRequestStatus& aStatus);
void SendTo(const TDesC8& aDesc, TSockAddr& anAddr, TUint flags,
            TRequestStatus& aStatus, TSockXfrLength& aLen);
void CancelSend();

```

```
void Write(const TDescC8& aDesc, TRequestStatus& aStatus);
void CancelWrite();
```

All these calls result in a call to the protocol to send data out onto the link. `Send()` can only be used with a connected socket (i.e., `Connect()` has been called successfully). `SendTo()` can only be used with connectionless sockets (see ‘Types of protocols’) as it provides the address of the remote peer to send the data to.

So when should you use each of these?

- Use `SendTo()` for connectionless sockets where there isn’t a fixed remote address to send the data to.
- Use `Send()` if you need to pass flags to the send call, or to find out how much data has actually been sent. The length of the sent data will always be the same as the length of the data sent unless non-blocking I/O has been set,¹ in which case the send call will send what it can without blocking and then complete immediately.
- Use `Write()` for stream sockets where you don’t need flags or to know how much data was sent.

The `aDesc` parameter is the data to be sent. This is an 8-bit descriptor as all protocols send byte streams rather than 16-bit UCS-2 data. The flags passed to `Send()` are protocol-specific, apart from `KSockUrgentWrite`. This flag can be used to mark the data for expedited sending, if the protocol supports urgent data (i.e., `KSIUrgentData` is set in the service information).

There are some other considerations when sending data which you should take into account:

1. Only one sending operation can be in progress at a time on a single `RSocket`, so you must wait for a `Send/SendTo/Write` to complete before calling any send operation again.
2. All send operations buffer the outbound data, so the data may not have actually been sent to the remote end by the time the call returns.

¹ ‘Blocking’ and ‘non-blocking’ I/O are something of a misnomer here – they refer to the operation that you’ve requested, rather than the state of your thread – i.e., a blocked operation will signal your `TRequestStatus` when the operation has completed. A non-blocking operation will complete the moment that a potentially long-running operation needs to take place with `KErrWouldBlock`. This confusion, unfortunately, arises from trying to layer blocking and non-blocking I/O (required to support `stdlib`) on top of asynchronous I/O (used in the rest of Symbian OS).

Our advice is to avoid the whole area of blocking and non-blocking I/O unless you’re porting software that needs to use it – by default sockets behave as you would expect them to, i.e., they complete your `TRequestStatus` when the operation has completed.

Of course, even if it has been there's no way to tell if the remote end has received it yet, so this isn't really a problem in practice.²

3. By default the send operation will block if a buffer isn't available for the data. This can be overridden by calling `RSocket::Ioctl()` with `KSoNonBlockingIO` (see 'Ioctls and options' below), but first read the previous advice on non-blocking I/O, i.e., this isn't actually a problem unless your application expects to get a response to its call within a 'short' time. Symbian OS applications using active objects for socket operations in conjunction with the socket API methods that take `TRequestStatus&` parameters can happily 'block' on a socket call as they can still do other useful work in the meantime.
4. For a datagram or sequenced packet socket, if the amount of data passed in the descriptor passed is larger than the maximum outbound data size then the send will be errored with `KErrTooBig`, and no data will be sent.

As you would expect, an outstanding `Send()` or `SendTo()` operation can be cancelled using `CancelSend()`, and an outstanding `Write()` operation can be cancelled using `CancelWrite()`. Of course, cancelling a send doesn't necessarily mean that the data won't be sent: the send processing by the protocol is asynchronous with respect to your program, so your program may have called cancel as the protocol is busily sending the data out onto the link, before it processes the cancellation request.

Receiving data Receiving data is done using the following methods:

```
void Recv(TDes8& aDesc, TUint flags, TRequestStatus& aStatus);
void Recv(TDes8& aDesc, TUint flags, TRequestStatus& aStatus,
          TSocketXfrLength& aLen);
void RecvOneOrMore(TDes8& aDesc, TUint flags,
                  TRequestStatus& aStatus, TSocketXfrLength& aLen);
void Read(TDes8& aDesc, TRequestStatus& aStatus);
void RecvFrom(TDes8& aDesc, TSocketAddr& anAddr, TUint flags,
              TRequestStatus& aStatus);
void RecvFrom(TDes8& aDesc, TSocketAddr& anAddr, TUint flags,
              TRequestStatus& aStatus, TSocketXfrLength& aLen);
void CancelRecv();
void CancelRead();
```

² We often see requests from developers asking for a way to find out if their data has reached the remote device yet. Our advice is always to incorporate a method into your protocol to do this. Even if it were possible to detect that the remote device has received the data (e.g., tracking acknowledged sequence numbers in connection-based protocols) that's no guarantee that the data has got to the application at the other end yet – it could just be sitting around in a buffer somewhere waiting to be read. And there's still no guarantee the remote system isn't going to crash before the application at that end reads it! So either accept that the service is best-effort from the moment you pass the data into the socket, or develop some form of response to the request you make, e.g., HTTP uses response codes to indicate that it has received and processed a request.

The first distinction is between `Recv()` and `RecvFrom()`. `Recv()` is used with connected sockets, and `RecvFrom()` with connectionless sockets. Otherwise they are identical, so let's focus on `Recv()` in this discussion.

`Recv()` provides the ability to receive a certain amount of data from the peer. The length of the data to be received is specified by the maximum length of the descriptor (*not* the current length of the descriptor). On completion the descriptor's length is set to the length of the data received.

Warning: if passing an `HBufC` to the `Recv()` method, be aware that the maximum length of the allocated `HBufC` may be greater than what was requested in the `NewL()` or `ReAllocL()`, which will cause `Recv()` to wait for more data than might be expected. To avoid this, wrap a `TPtr` of the right length over the `HBufC` and pass that to the `Recv()` call, or consider using `RBuf`.

As with the `Send()` calls above, there is the option to pass flags to modify how the `Recv()` is performed. The only generic option is `KSIPeekData` (if the protocol supports it), which returns the data but leaves the data in the socket's buffers, so the next read will return the same data. This can be useful if the incoming data consists of a known-length header with an unknown amount of extra data specified by a length field in the header. By first reading the header with `KSIPeekData`, the length of the extra data can be read out while leaving the header information in the socket buffers. Then you can provide a descriptor large enough to fit the whole message into and read the whole message including the header with a single `Recv()` call. This avoids the need to write code to manage reassembling the whole message from multiple `Recv()` calls.

The `Recv()` calls return when a packet is received for datagram or seqpacket sockets, or when the descriptor is full for stream sockets (see 'Types of sockets' above).

Warning: With a datagram or sequenced packet socket, you can lose data if the descriptor passed into `Recv()` is smaller than the incoming packet – the overflowing data is thrown away. This can be prevented by passing the `KSockReadContinuation` flag to `Recv()`, which modifies the behavior so that the next receive call will return the overflow data. See details in the Symbian OS Library for how this works.

This leads to a problem with stream sockets as it is impossible to know how much data the peer will send, so the `Recv()` call could block

indefinitely. There are two ways round this: using non-blocking I/O or using `RecvOneOrMore()`.

Non-blocking I/O is enabled by using an `ioctl` (see ‘`ioctls` and options’ in the next section), in which case the `Recv()` call will return `KErr-rWouldBlock` if there isn’t enough data available to fill the descriptor. Alternatively, `RecvOneOrMore()` will return immediately if there is data available, filling as much of the descriptor as there is data. If there isn’t any data waiting, the call blocks until data comes in.

Of the two, `RecvOneOrMore()` is more likely to be what you’ll need – with non-blocking I/O, if there’s no data waiting then the call returns immediately. There’s then no way to know when data has arrived from the remote peer without repeatedly calling `Recv()`, that is polling. Polling is inherently evil in a portable battery-powered device as it prevents the system going to lower power modes, and hence reduces battery life. Neither does the use of non-blocking I/O fit well with the standard Symbian asynchronous programming model, that is the use of active objects, which essentially provide non-blocking I/O via an alternate mechanism that also removes the need for polling.

Incidentally, the `TSockXfrLength` parameter is only useful when using the `KSockReadContinuation` flag. Otherwise, in all these calls the length of the descriptor is always set to the length of the data returned. In the case of using read continuations with a datagram or sequenced packet protocol, it indicates the number of bytes remaining in the current datagram, to allow an appropriately-sized buffer to be passed in on the next read to pick up the remaining part of the packet.

`Read()` provides a simplified interface to `Recv()` when you don’t need to pass flags.

An ongoing `Recv()` and `Read()` can be cancelled using `CancelRecv()` or `CancelRead()` respectively.

ioctls and options In addition to the send/recv calls for data transfer, the sockets API provides three ways to control the socket’s behavior and/or retrieve information:

```
TInt SetOpt(TUint anOptionName, TUint anOptionLevel,
            const TDesC8& anOption=TPtrC8(NULL, 0));
TInt SetOpt(TUint anOptionName, TUint anOptionLevel, TInt anOption);
TInt GetOpt(TUint anOptionName, TUint anOptionLevel, TDes8& anOption);
TInt GetOpt(TUint anOptionName, TUint anOptionLevel, TInt& anOption);
void Ioctl(TUint aCommand, TRequestStatus& aStatus, TDes8* aDesc=NULL,
           TUint aLevel=KLevelUnspecified);
```

`GetOpt()` / `SetOpt()` are used to synchronously read and write various socket options. There are some generic socket options implemented by ESOCK, and each protocol may provide additional protocol-specific options. For ESOCK options, the level is set to `KSOCKET`. For protocol-specific options the protocol defines various levels.

The generic options provided by ESOCK are shown in Table 3.3.

IOctls provide an asynchronous command channel to a socket that can be used for operations that may take a long time. ESOCK provides one generic ioctl: `KIOctlSelect`, which provides a functionality similar to the Unix `select()` call. Protocol-specific ioctls are documented in the Symbian OS Developer Library under the respective protocol. Outstanding ioctls can be cancelled using `CancelIoctl()`.

Disconnecting A connected socket can be disconnected using `Shutdown()`. While simply closing the socket will also disconnect any connection, the difference is that `Shutdown()` is asynchronous while `Close()` is synchronous. If the disconnection takes time, the `Close()` call will block until it is complete – this is unlikely to be what you want in your application. The `Shutdown()` methods are:

```
void Shutdown(TShutdown aHow, TRequestStatus& aStatus);
void Shutdown(TShutdown aHow, const TDesC8& aDisconnectDataOut,
              TDes8& aDisconnectDataIn, TRequestStatus& aStatus);
```

Note that connected socket here means that the protocol is a connection-oriented one (i.e., it doesn't have the `KSICongestionLess` flag set), it is not simply a socket on which `Connect()` has been called.

The `TShutdown` parameter specifies how the connection is to be closed. The options are:

`ENormal`

Cancel reads and writes, complete when remote end has confirmed close

`EStopInput`

Stop socket input and complete when output is stopped.

`EStopOutput`

Stop socket output and complete when input is stopped.

`EImmediate`

Stop socket input/output and complete (abortive close).

`EStopInput` will complete all outstanding reads with `KErrCancel` and block further reads, but will allow further writes to the socket. `EStopOutput` will do the opposite: cancelling writes and allowing reads. Both these options are only supported if the protocol has the `KSIGracefulClose` property.

Table 3.3 Generic options provided by ESOCK

Option name	Read/ Write	Meaning	Values
KSODebug	R/W	Debugging enabled or disabled	0 – Disabled 1 – Enabled Note that this value doesn't actually do anything!
KSORecvBuf	R/W	The receive buffer size used in ESOCK	KSocketBufSizeUndefined – sets to the default size 1 to KMaxTUInt: explicit buffer size Parameter passed as a TPckgBuf
KSOSendBuf	R/W	The send buffer size used in ESOCK	KSockBufSizeUndefined – sets to the default size 1 to KMaxTUInt: explicit buffer size Parameter packaged as a TPckgBuf
KSONonBlockingIO	R/W	Socket nonblocking mode	To set: no value needed To get: 0 – disabled, 1 – enabled
KSOBlockingIO	R/W	Socket blocking mode	To set: no value needed To get: 0 – disabled, 1 – enabled
KSOSelectPoll	R	Returns a bitmap of flags describing the status of the socket. Value is a TInt containing socket status flags	KSockSelectRead – data available to read KSockSelectWrite – writing to the socket is not currently blocked by flow control KSockSelectExcept – an error has occurred KSockSelectReadContinuation –

(continued overleaf)

(continued)

Option name	Read/ Write	Meaning	Values
			includes tail of prior read datagram as available data (i.e., indicates next read will be with read continuation)
KSOReadBytes Pending	R	Number of bytes available for reading	Value as TInt
KSOUrgentData Offset	R	Offset into the stream for urgent data (for protocols that support it)	TInt – offset in data for urgent data
KSOSelectLast Error	R	Last error on the socket	Error as a TInt
KSOMakeTransfer	W	Enables socket to be transferred to another process with given capabilities. This option must be called from the process owning the socket prior to a <code>RSocket : : Transfer ()</code> call being made by the receiving process	A <code>TSecurityPolicy</code> packaged as a <code>TSecurityPolicyBuf</code>
KSODisableTransfer	W	Disables the ability to transfer the socket	No value required

As with the `Connect ()`, if the protocol supports disconnect data (`KSIDDisconnectData`) then the version that supplies and receives that data can be used.

There is no way to cancel a shutdown once it has started, as the socket may already be in a half-closed state from which there is no way for the protocol to recover.

Once a socket has been shut down it cannot be used as all operations other than `Close ()` will return an error code. Don't forget that you still need to call `Close ()` on the `RSocket` instance to clean up the ESOCK client-server resources associated with the handle.

Process and thread sharing `RSocket` instances can be shared between different threads in a single process. Firstly the `RSocketServ` must be

set to shareable, using the `RSocketServ::ShareAuto()` function. Appropriate care must be taken to ensure that `SendRecv` operations are coordinated between the threads, as only one can be outstanding at a time on a socket. Patterns such as one thread reading and one thread writing are simple solutions. Care is also needed over closing the socket, as this should only be done once all threads have stopped using it.

ESOCK sessions cannot be shared between different processes, and therefore `RSocket` and the other ESOCK R-classes cannot be shared in this way. However, it is possible to transfer a `RSocket` from one process to another.

This is done using the `RSocket::Name()` and `RSocket::Transfer()` calls. The name of the socket to transfer is sent to the process that will receive the socket, and then the receiving process calls `RSocket::Transfer()` passing the `RSocketServ` instance to use and the name of the socket.

```
RSocketServ ss;
TInt err=ss.Connect();
// check err!
// name is in sockName variable
// must have been passed via other IPC mechanism
RSocket sock;
err=sock.Transfer(ss, sockName);
```

To be eligible for transfer, the source process must first set the socket as transferable, using a `SetOpt()`.

```
_LIT_SECURITY_POLICY_Cn(KProcPolicy, cap1, cap2, ...capn);
ret = destsock.SetOpt(KSOEnableTransfer, KSOLSocket,
    KProcPolicy().Package());
// check error!
```

Where `cap1`, `cap2...` are the capabilities that the destination process must have to effect the transfer. By using `_LIT_SECURITY_POLICY_Sn` or a `_LIT_SECURITY_POLICY_Vn` the transfer can also use a Secure ID or Vendor ID to ensure the recipient is a specific process, or a specific vendor's process.

Closing Closing down a socket is performed using the `Close()` method common to R-classes:

```
void Close();
```

There is a terminology issue here: `Close()` refers to closing the Symbian OS client-server connection (from your program to ESOCK)

rather than closing the connection to the remote peer, which is accomplished using `Shutdown()` (see above). However, as a side effect of closing the `RSocket` session handle, `ESOCK` and the protocol module will terminate the link to the peer. If the protocol supports graceful close then this behaves the same as performing a `Shutdown(ENormal)`, otherwise it's the same as calling `Shutdown(EImmediate)`.

There can be problems with graceful close of a socket when calling `Close()` without calling `Shutdown()` first. The problem occurs if more data is received after the application has called `Close()`. When this happens, the graceful close turns into an immediate close. This doesn't cause problems for your application, but you may see some non-graceful behavior on the wire, e.g., when using TCP, RST being sent. If you want to avoid this behavior, ensure you always use `Shutdown()` first.

3.1.5 Connection Management

One of the challenges of communications programming on smartphones is dealing with the multitude of connectivity options available. To address this complexity Symbian introduced the connection management APIs in v7.0s. As of v9.2, these APIs are largely used to manage IP network connections, and so more details are found in Chapter 6.

In addition to binding sockets to bearers, the connection management APIs provide mechanisms for enumerating, starting, stopping and monitoring connections, although again these are currently limited to IP-based connections.

RConnection

The core connection management API is provided by the `RConnection` class found in `<es_sock.h>`. The class provides two closely related services: the starting/stopping of connections and the monitoring of connections.

We'll look at the starting and stopping functionality first:

```
TInt Open(RSocketServ& aSocketServer,
          TUint aConnectionType = KConnectionTypeDefault);
TInt Open(RSocketServ& aSocketServer, TName& aName);
void Close();

void Start(TRequestStatus& aStatus);
void Start(TConnPref& aPref, TRequestStatus& aStatus);
TInt Start();
TInt Start(TConnPref& aPref);
TInt Stop();
TInt Stop(TConnStopType aStopType);
TInt Stop(TSubConnectionUniqueId aSubConnectionUniqueId);
TInt Stop(TSubConnectionUniqueId aSubConnectionUniqueId,
          TConnStopType aStopType);
```

RConnection objects exist in three states: without a connection to ESOCK (as is common for most server handles); as an unassociated connection and as an associated connection. The object starts without a connection to ESOCK, in a similar way to RSocket, and must be connected using one of the `Open()` calls. Once the client/server channel to ESOCK is established, then the RConnection object can be associated with a specific underlying bearer. Multiple RConnection objects can be associated with a single bearer connection, which allows multiple processes to monitor and control the bearers.

To start a bearer, one of the `Start()` methods is called. The simplest is `Start()` or `Start(TRequestStatus&)`. Both start the default bearer as configured on the device, the difference being the first is synchronous and the latter asynchronous. This may or may not result in the user being prompted, depending on how the device is configured. If you want more control over the bearer, use the overloads of `Start()` which take a `TConnPref` to allow specific bearers to be selected. `TConnPref` is a base class – you’ll always need to use one of the derived classes to actually specify what connection you want to open. There’s more details on this in Chapter 6.

Once the connection has been started, you should not attempt to `Stop()` it – despite it being the obvious operation to perform. `Stop()` really does mean stop, i.e., terminate the connection. `Close()` is the correct operation to perform on a connection when you’ve finished with it. In the case of IP connections, this will then shut down the network interface after an idle period defined in `CommsDat`.

Since a bearer can take a while to initialize and connect, it’s possible to retrieve notifications of the progress, for example to draw a progress bar on the screen. It is also possible to monitor the connection during its lifetime to watch for state changes. You may, for example, want to know if the connection goes down. Monitoring progress is one possible way to do that. Errors from socket operations are another. Since, for IP at least, errors are reported first via progress notifications, you may find this a better method.

The following methods are used to gather progress information:

```
void ProgressNotification(TNifProgressBuf& aProgress,
                        TRequestStatus& aStatus,
                        TUInt aSelectedProgress = KConnProgressDefault);

void ProgressNotification(TSubConnectionUniqueId aSubConnectionUniqueId,
                        TNifProgressBuf& aProgress, TRequestStatus& aStatus,
                        TUInt aSelectedProgress = KConnProgressDefault);
void CancelProgressNotification();
void CancelProgressNotification(TSubConnectionUniqueId
                                aSubConnectionUniqueId);
TInt Progress(TNifProgress& aProgress);
TInt Progress(TSubConnectionUniqueId aSubConnectionUniqueId,
              TNifProgress& aProgress);
TInt LastProgressError(TNifProgress& aProgress);
```

The remaining methods of `RConnection` can be used to inspect the state of a bearer. The bearer doesn't have to have been started by your program – it's possible to enumerate the active bearers and connect to one of these. This is done using:

```
TInt EnumerateConnections(TUInt& aCount);
TInt GetConnectionInfo(TUInt aIndex, TDes8& aConnectionInfo);
TInt Attach(const TDesC8& aConnectionInfo, TConnAttachType aAttachType);
```

`EnumerateConnections()` returns the number of active connections, and `GetConnectionInfo()` can then retrieve information about a specific connection. Don't worry about the number of active connections changing between `EnumerateConnections()` and `GetConnectionInfo()` – the server caches the information at the point when `EnumerateConnections()` is called, so that even if the connection is stopped it's still possible to retrieve the information. This does mean that there's no guarantee that a connection is still active when you get the information, but of course the bearer could be stopped by another process at any point anyway. `Attach()` can be called to bind your `RConnection` object to one of the connections you've discovered. Note that the `EnumerateConnections()` and `Attach()` operations are mainly useful for programs that display the current state of all connections to the user. Most programs should call `Start()` and let the normal selection process take place, even if asking for a specific IAP. This makes the code for starting a connection much simpler, as it doesn't matter whether the connection is started or not. In the case where the desired connection already exists, ESOCK and the rest of the framework will internally perform an `Attach()` for you.

The `aConnectionInfo` parameter to `GetConnectionInfo()` and `Attach()` is specified as a `TDesC8&`, but in fact the information returned is a `TPckgBuf<TConnectionInfoV2>`.

Now that your `RConnection` is associated with an active bearer, either through starting the bearer or via `Attach()`, you can retrieve various items of information about the connection.

```
TInt GetIntSetting(const TDesC& aSettingName, TUInt32& aValue);
TInt GetBoolSetting(const TDesC& aSettingName, TBool& aValue);
TInt GetDesSetting(const TDesC& aSettingName, TDes8& aValue);
TInt GetDesSetting(const TDesC& aSettingName, TDes16& aValue);
TInt GetLongDesSetting(const TDesC& aSettingName, TDes& aValue);

TInt Name(TName& aName);
```

The `Get...()` calls return information about the connection. `Name()` returns a name for the connection that can be used to bind other `RConnection` instances to the same connection using `RConnection::Open(RSocketServ&, TName&aName)`.

To allow programs to monitor the data flowing over a connection, there are various methods: `DataTransferredRequest()`, `DataSentNotificationRequest()` and `DataReceivedNotificationRequest()`. There's more information in Chapter 6 on these methods, or check out the documentation in the Symbian OS Library for details on how to use these.

One final oddity of `RConnection`: unlike nearly all R-classes in Symbian OS, it can't be bitwise-copied using `'='`. If you want to share an `RConnection` handle, you'll need a reference or pointer.

Subconnections Within the Symbian OS connection model, each `RConnection` can have multiple subconnections associated with it. A subconnection is used to configure the quality of service for sockets bound to the subconnection. This allows a single bearer to support different services with individual QoS requirements. For example an HSDPA packet data connection might have a VoIP session, a multiplayer game and a video download all occurring at the same time. Each service has specific QoS requirements: VoIP needs a guaranteed minimum bandwidth and low latency; the game needs low latency and the video download wants maximum bandwidth on a best-effort basis. By using the subconnection APIs, it is possible to associate each application's traffic with a separate PDP context with the required QoS negotiated and agreed with the network (again, see Chapter 6 for full details).

Most of the control of subconnections is done through the `RSubConnection` class (see below), but for historical reasons the `RConnection` class provides a basic ability to enumerate and query its subconnections.

```
TInt EnumerateSubConnections(TUint& aCount);
TInt GetSubConnectionInfo(TDes8& aSubConnectionInfo);
TInt GetSubConnectionInfo(TUint aIndex, TDes8& aSubConnectionInfo)
```

These functions work similarly to the connection enumeration functions. The descriptor parameter needs to be a packaged `TSubConnectionInfo`, created using `TPkgBuf<TSubConnectionInfo>` or `TPkg<TSubConnectionInfo>`. This API only provides very minimal information on a subconnection – use the `RSubConnection::GetParameters()` call to get more useful information.

For IP-based connections, the methods on `RConnection` and `RSubConnection` internally use different methods for gathering the information. As such it's possible that the `RConnection` values returned do not reflect the true number of subconnections. This will be addressed in future releases, but for now there is no reliable way to enumerate available subconnections. However, this shouldn't be a problem for most applications.

RSubConnection

RSubConnection provides the management interface to subconnections, allowing QoS parameters to be negotiated. There's more information on this in Chapter 6 and in the Symbian OS Library, so we won't go into the details here.

3.1.6 Naming and Lookup Services

We've looked at ESOCK's facilities for data transfer (*RSocket*) and connection management (*RConnection*), but there's one more challenge that ESOCK helps us overcome. That is figuring out how to contact the remote system we're interested in.

In all networks, devices (or more accurately interfaces) are identified by an address. Usually this is simply a number: for example the Ethernet adaptor on the computer I'm writing this on has the address 00-11-85-88-93-C5 (in hexadecimal), while the IP address is 10.18.212.1. Similarly, my Bluetooth hardware has an address of 00-0F-B3-4E-04-15. While numbers work fine for computers, they are distinctly unfriendly for humans.

As a result most types of network provide a human-readable name that can be translated to give the numerical address. The most common example is, of course, DNS names in IP networks, such as *www.symbian.com*, but other networks behave similarly. For example, Bluetooth provides a device name, such as 'Malcolm's P910', for each device. The major exception to this is infrared where the device to communicate with is usually identified by physical proximity. Device names are still available, but mainly used after a connection has taken place to indicate to the user the name of the device that has connected.³

Once the name has been resolved, the next challenge is to make connection with the right service. A single device can provide multiple different services: for example, a computer providing web, FTP, email and file-sharing services; or a phone providing file transfer and modem services. Each service appears on a different port which the client can connect to. In some systems, notably TCP/IP, the ports are well-known, fixed numbers e.g., HTTP on port 80, SMTP on port 25. In others, such as Bluetooth and IrDA, the port number is not necessarily fixed – remote devices support a service discovery database from which you can retrieve the necessary information required to create the connection.

ESOCK handles this address resolution and service discovery process through two classes, *RHostResolver* and *RNetDatabase*. There's

³ At this point we should note that device names provide no authentication of the remote machine. This should be obvious – anyone can set their Bluetooth device name to 'Malcolm's P910'. Even with DNS, it is possible to insert incorrect information into the system (search on the Internet for 'DNS cache poisoning' for an example of this). As a result, if you need to ensure that you really are communicating with the correct device, you need to use one of the security mechanisms described in the individual technology chapters, for example, TLS for IP networks, Bluetooth link authentication for Bluetooth.

another class confusingly called `RServiceResolver` which is obsolete and is not implemented by any protocol delivered by Symbian as part of Symbian OS. Some technologies provide wrappers over the `RSocket`, `RHostResolver` or `RNetDatabase` APIs to make them easier to use. See individual technology chapters for details of this.

RHostResolver

The `RHostResolver` class is at the heart of mapping names to addresses and vice versa. It provides a common interface across protocols, but since each protocol has its own method for address resolution, there are significant differences between the various protocol implementations. It's therefore important to consult the relevant chapter of this book to discover how to use this in practice.

Like all the ESOCK APIs, the first order of business is to connect to ESOCK:

```
TInt Open(RSocketServ& aSocketServer, TUint anAddrFamily, TUint aProtocol);
TInt Open(RSocketServ& aSocketServer, TUint anAddrFamily, TUint aProtocol,
          RConnection& aConnection);
```

Host resolvers are protocol-specific, so when you connect to ESOCK your program needs to specify which protocol you want to access. The flags are the same as for `RSocket`. You'll also notice that there's an overload of `Open()` that takes a `RConnection` parameter. In this case any communication that the protocol needs to do to satisfy requests will flow over the specified `RConnection`. This is necessary for example when two IP bearers are open, each with their own DNS servers.

```
void GetByName(const TDesC& aName, TNameEntry& aResult,
              TRequestStatus& aStatus);
TInt GetByName(const TDesC& aName, TNameEntry& aResult);
void Next(TNameEntry& aResult, TRequestStatus& aStatus);
TInt Next(TNameEntry& aResult);
```

The first functions provide a name to address lookup, taking a name and returning a `TNameEntry` which contains the name, address and any flags. If the protocol supports having a single name map to multiple addresses then you can call `Next()` to retrieve further addresses. `Next()` returns `KErrNotFound` when there are no more addresses for the name. Not all protocols support a name to address mapping, for example Bluetooth does not.


```
void GetByAddress(const TSocketAddr& anAddr,
                 TNameEntry& aResult, TRequestStatus& aStatus);
TInt GetByAddress(const TSocketAddr& anAddr, TNameEntry& aResult);
```

`GetByAddress()` performs the inverse mapping when possible – that is it takes an address and returns a name. This method is also used by Bluetooth to perform discovery of devices in range, but perhaps not in the way you’d initially expect – the ‘address’ in this case is a `TInquirySocketAddr` specifying the type of search you wish to perform. Bluetooth device addresses are returned from this search. More details on this subject can be found in Chapter 4.

RNetDatabase

`RNetDatabase` provides service discovery facilities. These are highly protocol-specific, so we’ll defer discussion of where and when to use this to the chapters dealing with each protocol. Do not be surprised if you do not see explicit usage of `RNetDatabase` in those chapters though, as it is often concealed behind a set of utility classes – for example, `CSdpAgent` in the case of Bluetooth.

3.2 Into Practice

Now it’s time to roll up our sleeves and try putting some of this theory into practice. We’ll start with a very simple program to retrieve the Symbian homepage to show the basics. Then we’ll build a simple wrapper around `RSocket` that you can use as the basis for your own sockets programming. The focus will be on `RSocket` as it’s at the heart of accessing resources on networks. The connection management and resolution services are covered in more detail in later chapters.

As web pages are served using the HTTP protocol, which is transported over TCP/IP, this example will touch on some TCP/IP networking concepts. These are more fully explained later in the book and so we won’t be giving much detail here. This is an extremely low-level way to access the HTTP protocol – the Symbian HTTP framework (described in Chapter 11) provides a better method.

To get any of these examples to work on the Symbian emulator you’ll need to have configured the emulator to access an internet connection – see Chapter 13 for details on how to do this.

3.2.1 The Bare Bones

Our simple example program can be found in the `esock\simple1` directory. If you look in the `simple1.cpp` file at the `getPage()` function, you’ll see code to retrieve the page.

First, let's see the class declaration:

```
class CGetWebPage : public CActive
{
    enum TState
    {
        EConnectingSocket,
        ESendingData,
        EReceivingData
    };
...
public:
    static CGetWebPage* NewL(); // the usual pattern
    void GetPageL();
    virtual void RunL();
    // we would also need to implement DoCancel() in real code
...
private:
    void ConstructL()
private:
    RSocketServ iSs; // needs Connect()ing, but omitted for clarity in this //example
    RSocket iSock;
    TInetAddr iWebServAddr;
    RTest iTest;
    TBuf8<255> iData;
    TBuf16<255> iData16; // purely for display purposes using iTest
    TSockXfrLength iLen;
}
```

As you'd expect, the first thing the code does is connect to the socket server, and open a socket.

```
void CGetWebPage::ConstructL()
{
    // Connect to the socket server
    User::LeaveIfError(iSs.Connect());

    // Open the socket
    User::LeaveIfError(iSock.Open(ss, KAfInet, KSockStream,
                                KProtocolInetTcp));
}
```

As we're using TCP/IP for this example, we specify the TCP/IP protocol in the parameters to `RSocket::Open()`:

- `KAfInet` – the internet protocol family
- `KSockStream` – TCP is a stream protocol
- `KProtocolInetTcp` – the TCP protocol.

The same code works for other protocols such as Bluetooth or IrDA – simply change the parameters to the `Open()` call.

Next we will connect to the web server at *www.symbian.com*. In a real program we would use the DNS facilities of Symbian OS to look up the right IP address (see Chapter 6 for details), but here we've hard-coded it. Don't do this in a real program – IP addresses can and do change! We also hardwire the port number as 80, but that's OK as this is the well-known port for web servers and so won't change.

```
void GetPageL()
{
    iState = EConnectingSocket;
    iAddr.SetAddress(INET_ADDR(81,89,143,203)); // hardwired address of
                                                // www.symbian.com - DON'T do this in real code
    iAddr.SetPort(80); // HTTP port
    ...
}
```

`TInetAddr` is the TCP/IP-specific address class derived from `TSockAddr` so can be passed to the `RSocket::Connect()` call. Next we make the connection, then send the HTTP request to the web server. Again, this has been hard-coded here rather than being built up out of the individual HTTP request headers.

```
_LIT8(KReq, "GET / HTTP/1.0\r\nHost: www.symbian.com\r\n\r\n");

void CGetWebPage::GetPageL()
{
    ...
    User::LeaveIfError(iSock.Connect(iAddr, iStatus));
    SetActive();
}

void CWebPage::RunL()
{
    switch(iState)
    {
        case EConnectingSocket:
            iTest.Printf(_L("Connect returned %d\n"), iStatus.Int());
            iTest.Printf(_L("Sending request...\n"));
            iState = ESendingData;
            iSock.Send(KReq, 0, iStatus);
            SetActive();
            return;
        case ESendingData:
            ...
    }
}
```

Provided the send completes successfully, we can now read the response from the web server:

```
class CGetWebPage : public CActive
{
    ...
}
```

```

void CWebPage::RunL()
{
    ...
    case ESendingData:
        iTest.Printf(_L("Request sent\n"));
        iTest.Printf(_L("Receiving data...\n"));
        iSock.RecvOneOrMore(iData, 0, iStatus, iLen);
        iState = EReceivingData;
        SetActive();
        return;

    case EReceivingData:
        iTest.Printf(_L("Recv status %d, len %d\n"), iStatus.Int(),
            iLen());
        iData16.Copy(iData);
        iTest.Printf(iData16);
        break;
    ...
}

```

Notice that we use `RecvOneOrMore()` to receive the response, as we don't know how long it will be. If we had instead used `Read()` with the same buffer, if the response was less than 255 bytes the program would have hung waiting for more data from the web server.

That's it – you've now seen the simplest program which communicates over TCP/IP. As you can see, this already does something useful, in this case retrieve a web page. This is often the case when creating a client program – relatively little code can result in big effects, since the server already exists. Of course if you're having to write both the client and server there's no free lunch and you'll have plenty to do!

From this simple beginning we'll now move on to a set of skeleton socket wrapper classes that deal with most of the housekeeping needed for socket communications. To keep the example clear, we have made some design tradeoffs:

1. The reading and writing are targeted towards stream sockets. The code will work with datagram sockets, but any datagram boundaries will be lost.
2. There's no buffering strategy implemented. How to manage the buffers is left to the caller. Depending on the final application, buffering may be critical to performance. For example, a program which is downloading a lot of data to a file will probably want to use a double-buffering strategy to maximize throughput, whereas a program which needs low-latency transmission of small amounts of data would need to minimize the use of buffers.
3. The classes provide a call-back based API for event notification. This means that internally they provide active objects to interface to the asynchronous calls and then transform the completed events into

callbacks. We have taken this approach to show how to write the active objects needed to use RSocket.

The code for this wrapper can be found in `\esock\csocket`. It will be useful to have the code handy while reading this section, so you can trace through the discussion, however, the most important code snippets are shown here so don't worry if you don't have access to the code right now.

There are two classes at the heart of this wrapper: `CSocket` and `CSocketFountain`. Let's start with `CSocketFountain`. You use `CSocketFountain` if you want to receive incoming connections. It provides a simple way to start listening for connections, and when one comes in it passes a new `CSocket` instance back, ready to be used to communicate with the remote peer. Hence the name – you can think of this class as creating a fountain of new sockets, one per connection. Just like a fountain there is a catch – there's no way to rate-limit how many connections are created, so it's possible to get soaked! If your program is going to be running on a network where large numbers of connections might be made, either intentionally or maliciously (i.e., the Internet), then you should add some form of rate-limiting on incoming connections and a cap on the maximum number of simultaneous connections to prevent denial of service attacks on the device.

`CSocket` provides methods to make an active connection, and, once connected, methods for sending, receiving and closing down the socket.

Let's look at how `CSocket` and `CSocketFountain` are used:

```
TInetAddr addr(KInetAddrLoop, 7777);

// create a listening connection
iSockFountain = CSocketFountain::NewL(*this);
iSockFountain->Listen(KAfinet, KSockStream, KProtocolInetTcp, addr);

// then create a connection to it
iSock = CSocket::NewL();
iSock->SetSocketCallback(*this);
iSock->Connect(KAfinet, KSockStream, KProtocolInetTcp, addr);
```

First we create a new `CSocket`, and set the socket to callback the appropriate object (in this case `*this`). The callback is via a `MCSocketCallbacks` interface, so the caller must implement this interface. Next we setup a `CSocketFountain`, here the callback is through the `MIncomingConnection` interface which again we must implement. Finally, we tell the socket fountain to listen on a certain address, and then the `CSocket` to connect to the same address. The address we're using is the localhost address, so the result is that the `CSocket` is looped-back to our `CSocketFountain`. Of course this would also work using a remote address and having two devices both running the program.

When `CSocketFountain` receives a connection, a new `CSocket` is created and passed back to the fountain owner, through the `MIncomingConnection` interface:

```
class MIncomingConnection
{
public:
    virtual void IncomingConnection(CSocketFountain&, CSocket& aSock)=0; //
                                                // Transfers ownership
    virtual void ListeningError(CSocketFountain& aFountain, TInt aErr)=0;
};
```

Both callbacks specify the `CSocketFountain`, to make it easier to keep track when using multiple `CSocketFountain` objects within a class, e.g., when listening on multiple ports or addresses.

That's it on the receiving connections side – the `CSocket` passed to `IncomingConnection()` is ready for use.

On the `CSocket` side, once the connection is made, a callback occurs through the `MCSocketCallbacks` class:

```
class MCSocketCallbacks
{
public:
    virtual void ConnectComplete(CSocket&, TInt aErr) = 0;
    virtual void NewData(CSocket&, TDes8& aBuf) =0;
    virtual void WriteComplete(CSocket&, TInt aErr, TInt aLen) =0;
};
```

As with `CSocketFountain`, the `CSocket` that is making the callback is passed as an argument, so that the client can have one callback function to handle multiple `CSockets`. This is good practice in any callback interface – always identify the source of the callback, so that a single processing function can handle multiple objects without having to have its own way to track the ownership.

Now let's take a look inside the implementations of `CSocketFountain` and `CSocket`.

CSocketFountain

Here's the main methods for `CSocketFountain`:

```
class CSocketFountain : public CActive
{
public:
    static CSocketFountain* NewL();
    static CSocketFountain* NewL(MIncomingConnection& aCallBack);
    static CSocketFountain* NewL(RSocketServ& aSS);
    void SetConnectionCallBack(MIncomingConnection& aClient);
    void Listen(TUint addrFamily, TUint aSockType, TUint aProtocol,
               TSockAddr& aAddr);
```

```
void StopListening();
...
};
```

We've provided several overrides for `NewL()` to allow various combinations of external state to be provided. If nothing is supplied then `CSocketFountain` creates its own connection to `ESOCK`, but it is more efficient to share a `RSocketServ` instance between all the objects in a thread, so an `RSocketServ` connection can be passed in.

We have also allowed for the callback destination to be changed after construction. This is a useful technique, as it allows flexibility for the user in construction order (if the eventual destination isn't constructed before the `CSocketFountain`), and also for the callback to change as the program's state changes.

The heart of the API is the `Listen()` call, which calls `ListenL()`:

```
void CSocketFountain::ListenL(TUint addrFamily, TUint aSockType,
                             TUint aProtocol, TSockAddr& aAddr)
{
    iClosing = EFalse;
    User::LeaveIfError(iListenSock.Open(iSS, addrFamily, aSockType,
                                       aProtocol));
    // Now start listening
    User::LeaveIfError(iListenSock.Bind(aAddr));
    User::LeaveIfError(iListenSock.Listen(4)); // Q of 4
    IssueAcceptL();
}
```

This is surprisingly simple – first we use the `Open()` call, then I call `Bind()`, then `Listen()` to set the local listening address and start listening. Finally `IssueAcceptL()` is called to get the first incoming connection. This is broken out as a separate function as it will also be called from the `RunL()` each time an accept completes, to receive another incoming connection.

`IssueAcceptL()` checks to see if the fountain is closing down, and if not calls `RSocket::Accept()` passing a new blank socket.

Finally, the `RunL()` calls back to the fountain's owner and issues another `Accept()` call. All the error handling for the `CSocketFountain` is done through a `RunError()` function, which allows the `RunL` to leave if there are any problems. The error handling is fairly simple – the error is reported to the client, and the listening socket closed. The fountain can be reused by calling `Listen()` again.

Now let's look at the outbound half of the equation – `CSocket` and active connections.

3.2.2 CSocket Active Connections

`CSocket` active connections are started through the `Connect()` call.

```

EXPORT_C TInt CSocket::Connect(TUint addrFamily, TUint aSockType,
                               TUint aProtocol, TSocketAddr& aDest)
{
    // Open the socket
    TInt err = KErrNone;
    if(iUseConn)
    {
        err = iSock.Open(iSS, addrFamily, aSockType, aProtocol, *iConn);
    }
    else if(iUseSubConn)
    {
        err = iSock.Open(iSS, addrFamily, aSockType, aProtocol, iSubConn);
    }
    else
    {
        err = iSock.Open(iSS, addrFamily, aSockType, aProtocol);
    }

    if(err == KErrNone)
    {
        // Now start the connection
        iSock.Connect(aDest, iStatus);
        SetActive();
    }

    return err;
}

```

This starts with some logic to open the socket in the right way depending on whether a connection or subconnection was supplied when the object was created, and then calls `RSocket::Connect()` with the supplied address. When the connection completes, `CSocket's RunL()` will be called, which simply calls back to the owner to inform them. At this point the `CSocket` is ready for sending and receiving.

Sending and receiving

Sending data is relatively simple – a buffer is passed in to `CSocket::Send()` which then calls `RSocket::Send()`. When the send completes, a callback occurs through `MCSocketCallback::WriteComplete()`. This is done through the `CSocketWriter` class, which is a small active object used to handle the asynchronous `Send()`. As we mentioned previously, be aware that `Send()` completing only means that the data has been copied to the protocol's internal buffers, not that it's actually been sent! The callback also signifies that the client is free to modify the data in the descriptor passed to the `Send()` call – whilst the `Send()` call is in progress the contents of the descriptor must remain untouched, as the protocol may access it at any time.

One problem with communications and active objects is the scheduling between the sender and receiver active objects. Ideally we'd want them to each have about half the time, and for neither of them to be

starved if there's a lot of data moving in or out. However, if two active objects at the same priority are ready to run then the active scheduler will always schedule the one that was added to the scheduler first. This can easily result in starvation. The solution used here is to remove the `CSocketWriter` (and `CSocketReader`) from the active scheduler each time the `RunL()` fires, and then re-add it. This has the effect of moving the active object to the back of the queue and thus allowing the other one to run, even if the next call completes immediately.

Receiving data is slightly more complicated due to the differences in semantics for `Recv()` and `RecvOneOrMore()` between stream and datagram protocols. The approach used here is to always provide `RecvOneOrMore()` semantics for stream sockets since this is almost always what is wanted. For datagram sockets the code additionally prevents data loss from passing a too-small buffer by passing the `KSockReadContinuation` flag to the `Recv()` call. The side effect of this is that datagram boundaries may be lost if the buffer provided is smaller than the incoming datagrams, but in general this approach works well.

`KSockReadContinuation`

Here's a practical example of the `KSockReadContinuation` flag we discussed earlier. With a datagram protocol, the `RSocket` receive calls return a *whole* datagram at a time. If the buffer provided is smaller than the amount of data in the datagram, the excess is silently dropped.

If that's not the behavior we want, then the `KSockReadContinuation` flag can be passed to the `Recv()` call. This instructs ESOCK to hold onto any overflow, and return it in the next call to `Recv()`. If the `Recv()` call provided a `TSockXfrLength` parameter then this is used to indicate the length of the remaining data.

Here's an example where the remote device sends a 250-byte datagram:

```
TBuf8<100> iBuf;
...
sock.Recv(iBuf, KSockReadContinuation, iStatus, iXfrLen);
// completes w/ KErrNone, iBuf contains bytes 0-99, iXfrLen 150
// (remainder of first datagram)
...
sock.Recv(iBuf, KSockReadContinuation, iStatus, iXfrLen);
// completes w/ KErrNone, iBuf contains bytes 100-199, iXfrLen 50
// (50 bytes remaining in first datagram)
...
```

As with the `CSocketWriter` class, the `RunL()` dequeues and requeues the active object to ensure fair scheduling.

Buffering

While the code presented here doesn't provide any buffering strategy, it's something that's worth thinking about carefully for your application as it can have a major impact on performance.

First some background. Communications links have four fundamental performance measures: bandwidth, latency, jitter and loss. Bandwidth is the amount of data per unit time (e.g., 10 Mb/s). Latency is the time taken for the data to transit the network from sender to receiver (e.g., as we write this, the time for a packet to reach *www.google.com* from the machine is approximately 15 ms). Jitter is the variation in packet arrival times given a constant transmission rate, i.e., the variance in latency. Loss (or packet loss) is the number of packets sent that don't make it to the receiver (often expressed as a percentage).

These four quantities are interrelated. The available bandwidth puts a lower limit on latency – if you have 2400 bits/s bandwidth then the minimum latency for a single bit is $1/2400$ s or 0.4 ms. Jitter can be smoothed by buffering, but this increases latency as packets wait in the buffers. And packet loss impacts bandwidth, jitter and latency.

In applications that perform some 'real-time' service it is latency or jitter rather than absolute bandwidth that is important. For example, a car carrying 100 DVDs between two points one hour apart has a bandwidth of 400 GB/hour or 113 MB/s (~ 1 Gb/s) but with a latency of one hour, which would make for a very painful user experience for anything interactive, while being ideal for non-real-time services like transferring software.

For your specific application, you'll need to make design choices to trade-off these four factors to get the performance you need. The first and simplest choice is packet loss. If you can tolerate high jitter and latency but not losing a single bit, then selecting a reliable transport (e.g., TCP) will give this. In some cases, rather than selecting a different transport protocol, you might just configure it in a different way, e.g., L2CAP can be configured to provide a reliable transport (the default), or to discard data if it hasn't been transmitted within a certain period.

For jitter, adding buffering at the receiver will smooth out the flow of data, which is useful for any process that wants to consume data at a fixed rate (e.g., media playback). The flip side of this is that latency will increase, so for interactive communications the buffers mustn't be too big.

Latency is largely a feature of the network and outside your direct control, but can be improved by sending data in smaller packets. This allows urgent data to be sent faster as it doesn't have to wait for large outgoing packets to clear, however, this often increases the protocol overhead, as the amount of data in each packet is smaller whilst the packet header length stays constant. So an improvement in latency is matched by a reduction in (user-visible) bandwidth.

User-visible bandwidth can be improved by keeping the network fully utilized. To do this, it's important to start a new send or receive operation as soon as the previous operation completes. If your program needs to process the incoming data, then a double-buffering strategy is often useful, where one buffer is being filled by ESOCK whilst you process the other, then, once both sides have finished processing their current buffers, you can switch them over. This can result in substantial performance gains.

Closing down

The normal way to close a `CSocket` is simply to delete the object. This automatically cleans up and cancels any outstanding reads or writes and shuts any ESOCK handles.

If you need the shutdown behaviors provided by `RSocket::Shutdown()` then you can use the `CSocket::Socket()` call to retrieve the underlying `RSocket` and call `Shutdown()` on this, before deleting the `CSocket`.

It's not possible to reconnect a `CSocket` after it's been disconnected as not all protocols support this. Rather than code round this, it is simpler to just delete and recreate the `CSocket` object.

3.3 Summary

In this chapter we've covered:

- an overview of ESOCK's services
- the different types of socket – stream, sequenced packet and datagram
- the differences between connection-oriented and connectionless protocols
- the main ESOCK API classes: `RSocketServ`, `RSocket`, `RHostResolver`, `RNetDatabase`, `RConnection` and `RSubConnection`
- how to use the sockets API to make connections and to receive incoming connections
- sending and receiving data over a socket
- a basic framework for writing your own socket-based communications programs.

4

Bluetooth

This chapter starts with a description of Bluetooth technology. Even if you have some experience with Bluetooth it may still be worth skimming through the technology part of the chapter just to understand how all the parts fit together.

Each section in the technology part has a corresponding section in the description of how Bluetooth is implemented in Symbian OS. Thus you can rapidly move between concepts and implementation to understand how Symbian OS is presenting the underlying Bluetooth functionality.

There is a lot of detail in this chapter as we cover many different areas of Bluetooth. For building basic Bluetooth services, we recommend you read:

- discovering and connecting to devices (sections 4.1.2 and 4.2.2)
- service discovery and SDP (sections 4.1.6 and 4.2.8)
- L2CAP (sections 4.1.5 and 4.2.5).

4.1 Bluetooth Technology Overview

Bluetooth is a general-purpose short-range wireless technology, which presently operates in the industrial, scientific and medical band at 2.4 GHz, although in future it may run over other radio bands. It is designed with resource-constrained devices in mind. In particular, Bluetooth optimizes battery consumption – a key issue with mobile devices. Originally Bluetooth was seen as replacing two common cables – the RS-232 serial cable and the cable linking a mobile phone and a headset. Increasingly, Bluetooth is now replacing other cables: stereo audio, video, parallel, USB and Ethernet.

One of the key axioms of Bluetooth is to assume that devices have a similar underlying level of functionality,¹ rather than assuming one ‘smart’ host and many, relatively ‘dumb’ devices – contrast this with USB, which was designed with the ‘one host, many peripherals’ approach. This is now being rectified in USB with the On-The-Go (OTG) standard; however, it requires additional implementation in all devices that wish to support it.

One benefit of the more symmetric Bluetooth approach is that a Bluetooth connection can be initiated by either device, and used for any purpose – allowing use cases, such as peer-to-peer messaging, to be supported that would be much harder to implement and configure using more host-centric technologies.

Another major strength of Bluetooth is the support for ‘profiles’. These consider end-to-end use cases, such as connecting a headset to a mobile phone, and specify standard ways of achieving them. The benefit of this is that a Bluetooth device does not require individual drivers to perform a task: if it did, then all Bluetooth devices would need some store of drivers for each other potential Bluetooth device with which they wished to communicate. The profile concept is similar to the USB concept of device class, although typically Bluetooth profiles go further in describing a use case than the USB device class specifications.

Bluetooth profiles also provide a benefit over technologies such as WLAN, which concentrates on providing a wireless version of Ethernet technology only – higher level specifications are set by other standards groups, such as the IETF. As a result, assembling the stack of protocols and defining a configuration required to implement a use case can be far harder when using WLAN technologies, and equipment using different technologies for implementing the same use case can be on the market at the same time – leading to user confusion. The wide range of devices enabling voice-over-IP is evidence of this – some use SIP, some use proprietary protocols, and all are forced to interoperate using the lowest common denominator of the PSTN.

The Bluetooth standards comprise a number of protocols and profiles – currently around 8 and 12, respectively – that describe how to use the protocols in an interoperable way to fulfil various use cases such as printing, networking, exchanging files, listening to audio, watching video and connecting mice and keyboards.

¹ By this we mean there is nothing in the Bluetooth specifications that requires one device always to act as a service provider. Instead any device could potentially act in any role – this is purely a function of the software support above the L2CAP layer. So, for example, a single mobile phone could simultaneously be acting as a modem for a connected laptop, and utilizing the services of a Bluetooth headset for making a phone call. Such an arrangement would be extremely difficult using USB (aside from the tangle of wires it would cause!).

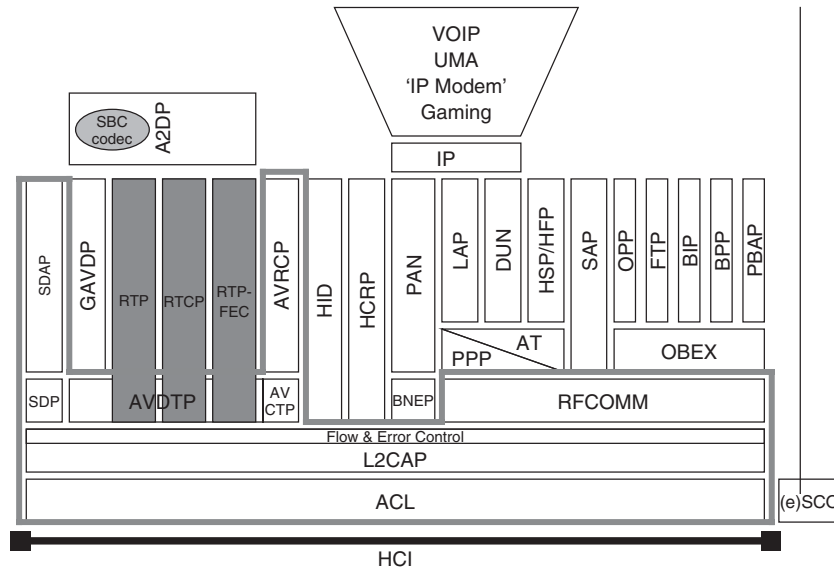


Figure 4.1 Bluetooth protocols, indicating those discussed in this chapter

Figure 4.1 shows the profiles and protocols that make up the Bluetooth specifications – the box indicates the areas that are discussed in this chapter. Bluetooth is such a wide-ranging technology that it would be impossible to cover all areas, especially as some of the profiles are not of particular interest to developers: often they are implemented entirely within the OS and do not present any user-visible APIs – PAN, HID and A2DP are examples of such profiles which are hidden behind existing general-purpose APIs.

4.1.1 Architectural Overview

Bluetooth devices are usually implemented in two parts: the host and the controller. Effectively, the controller consists of the Bluetooth hardware along with the lowest levels of the Bluetooth stack such as the physical link manager; and the host is a set of higher level protocols and profiles and other supporting libraries that interact with the controller. The host and the controller communicate via the host controller interface, HCI (which may not be physically present, but always exists in Symbian OS devices, and is worth at least modelling logically). The controller lies under the HCI in Figure 4.1 but is not shown as it is discussed only briefly in this chapter because all functionality is accessed through the higher level Symbian OS APIs.

Within the controller, a number of entities control the power of the radio, the frequencies on which it transmits and receives at any particular

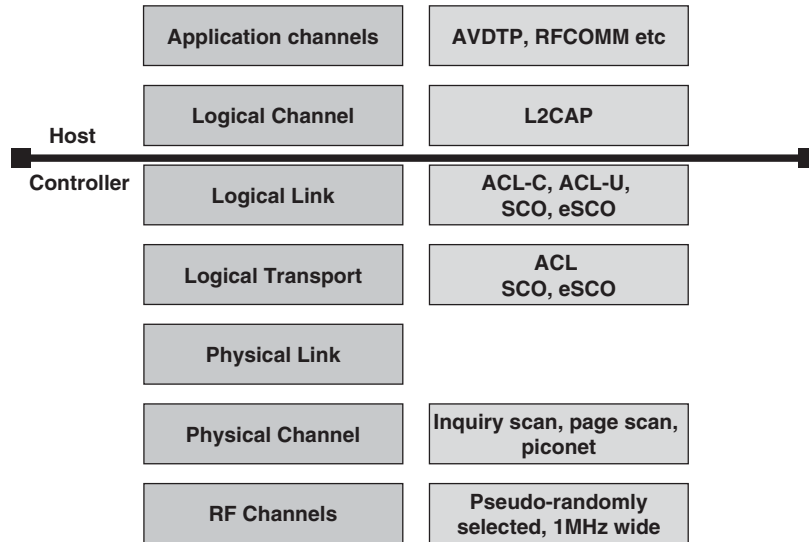


Figure 4.2 Bluetooth channel architecture

time, communication with the host, and direct communication with the peers in other Bluetooth hardware. Figure 4.2 introduces the types of transport entities at the lowest layers in Bluetooth; and shows where the host/controller split lies. The left-hand side of the diagram is the class of transport, the details of which we'll explore later in the chapter, the right-hand side enumerates the instances available of that type – again, we'll discuss these in more detail later.

Most of the protocols are implemented within the host, along with the profiles. The lowest level of the host protocols is L2CAP, which presents logical channels to higher level protocols, applications and profiles. L2CAP is roughly analogous to a hybrid of TCP and UDP in terms of functionality. Unlike TCP and UDP, where the requirement for reliability vs. timely delivery forces you to choose between the two protocols, L2CAP, when operating with the optional flow-and-error control feature, can be configured to choose between the two options within the same protocol.

4.1.2 Discovering and Connecting to Devices

Wireless devices must provide some 'association model' to allow the user to select other devices with which to interact. For wired devices the model involves physical interaction with the other device, typically the insertion of a cable. In the wireless case there is obviously no cable involved, so what is the alternative?

The usual method is to make the local device 'visible' when it wishes to be discovered by other devices. Devices are usually visible via the

technology that will ultimately be used to exchange data with them, i.e., visibility is handled ‘in-band’.²

In Bluetooth, a number of concepts are used when referring to the state of two devices that require association:

- discoverable
- connectable
- pairable.

Discoverability

If a device is discoverable it will respond to searches made by other Bluetooth devices. Therefore other devices which are actively scanning will be able to discover it.

There are actually two types of ‘discoverability’ – limited and general. If a device is in ‘limited discoverable’ mode then it can be found by devices performing ‘limited’ or ‘general’ discoveries. If a device performs a ‘limited’ discovery, it will find *only* devices in ‘limited discoverable’ mode.

Devices should only stay in limited discoverable mode for a restricted period of time: the corollary of which is that the user of the device being discovered would have to perform some specific action in the UI to activate the mode; typically when they are expecting some Bluetooth activity – for example, if they are preparing to receive a vCard via Bluetooth.

The difference between limited and general discoveries is that the devices taking part in the discovery procedure are using a different ‘inquiry access code’ (IAC) in the Bluetooth hardware.

Connectability

Connectability is somewhat orthogonal to discoverability within the Bluetooth specification. If a device can receive a Bluetooth physical connection then it is said to be ‘connectable’. Note that a device need not be discoverable (in either the limited or general forms) to be connectable. However, it rarely makes sense for a device to be discoverable but *not* connectable.

Pairability

As part of the process of forming a connection to a service on a remote Bluetooth device, the devices may wish to authenticate each other and form a long-term security relationship. This allows the devices to authenticate each other in future without further input from the user.

² Consider the alternative ‘out-of-band’ method, where, for example, devices could be discovered and paired using another technology, such as some form of embedded RFID chip, by touching them together.

The initial authentication process requires that the same ‘passkey’ be entered into each device when prompted by the UI. The passkey will ultimately yield a security token (the ‘link key’) shared between the two devices. If a long-term security relationship is to be established this link key will be retained in persistent storage.

A Bluetooth device is said to be ‘pairable’ if it allows the input of a passkey.

Class of device

When a device responds to discovery, it sends a three-byte value called the ‘Class of Device’, or ‘CoD’. In the present specification this comprises two fields describing the broad nature of the Bluetooth device and a third field which outlines the services it provides. The first two fields are the ‘major’ and ‘minor’ device class, and the third is the ‘major service class’.

The major and minor device classes are often collectively referred to (perhaps unhelpfully!) as the ‘device class’, and describe, in loose terms, the physical appearance of the device: for example whether the device is a computer, a phone, a printer or a headset.

The service class (since there is no ‘minor’ service class, the ‘major’ is usually dropped – but care is needed to distinguish this from ‘service class’ in SDP, which is discussed later) is a bit field with individual bits representing – at a high level – services available on the device at the time it responded, such as ‘networking’, ‘rendering’ and ‘object transfer’. Major service class only provides hints as to the services available, especially as more than one service may map onto a single service class bit – for example, both Bluetooth A2DP headsets and Bluetooth printers would set the ‘rendering’ bit: for conclusive results as to the availability of services on a Bluetooth device, the Service Discovery Protocol (SDP) should be used. See section 4.1.6 for more on SDP.

The service class bits are shown in Table 4.1.

The semantics of each are quite loose, and generally up to the application to choose, or a profile specification to advise, of the appropriate bits to set for a given service.

The class of device information for a remote device is also obtained by the Bluetooth stack when it receives an inbound connection. Unfortunately there is no other way to ascertain the class of device of a remote device than during the inquiry phase, or at the time of an inbound connection at the physical link level (physical links are discussed in more detail later). This means that in certain situations, such as a new service starting on an already-connected device, the current class of device information for a remote device may not be known.

4.1.3 Physical Links

Once a device has discovered another device, it may form a physical link between them. The device creating the connection is, by default,

Table 4.1 Class of device service classes

Service class bit	Example of use
Limited discoverable mode	This bit is not really a 'service class', despite it being in the service class field of the class of device. Applications cannot control this
Positioning	Bluetooth GPS device providing position coordinates
Networking	Bluetooth modem providing access to a network
Rendering	Bluetooth printer
Capturing	Bluetooth web-cam that captures images and forwards them to a display
Object transfer	A Bluetooth device that implements OBEX
Audio	Bluetooth headset
Telephony	Bluetooth-enabled mobile phone
Information	Bluetooth-enabled advertising display

the master of that link; the receiver of the connection is, by default, the slave. At most one physical link exists between a master device and a slave device in a piconet (see Figure 4.3). The master and slave roles can be swapped. However, physical links always emanate from a master – Bluetooth is a star topology.

A piconet is formed when a master is connected with at least one slave; all slaves connected to a single master are coordinating their use of the RF channels so they are all tuned to the same RF channel at the same time (which changes, in a piconet, 1600 times per second). The piconet is one type of 'physical channel'; others are used when a device is waiting to be discovered, and another is used when a device is expecting a physical link.

It is also worth noting that slaves in a piconet cannot communicate with each other – each slave can only connect to the master device. The only way for slaves in a piconet to communicate is using a higher level service, such as the virtual Ethernet network provided by PAN profile.

Two different piconets can be brought together to form a scatternet. No direct Bluetooth connection is made between the two piconets; all that happens is that a device can switch its radio between two frequency-hopping sequences, allowing it to operate in two piconets simultaneously. By definition, it is *not* possible, using one Bluetooth radio, to be a master in one piconet whilst simultaneously being a master in another piconet. It is possible for a device to be a master of one piconet and a slave in

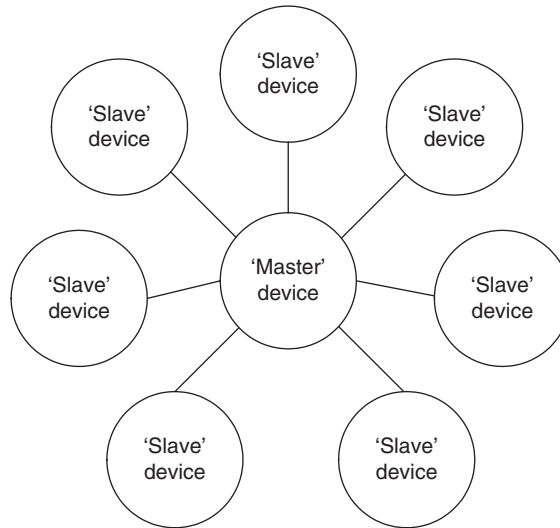


Figure 4.3 Bluetooth piconet topology

another; however, due to the time division multiplexing, the bandwidth available to the device in each piconet can become unacceptable. It is also – just – possible for a device to be a slave in two piconets. The ability to participate in multiple piconets is entirely dependant on the capabilities of the controller – the only time the host knows about this is either when explicitly examining and manipulating the piconet roles, or when link creation fails because the controller cannot meet the master/slave requirements of that new connection.³

The practical consequence of this is that it is not always possible to connect to a device that is in range, as it may be participating in other piconets and not be able to accept another connection. In these cases, it is best just to inform the user the remote device is busy rather than try and manipulate roles in order to achieve a connection, especially as no single device has enough information to work out if there is a suitable set of roles that will allow every other device involved to connect to each other in the desired topology.

The properties of the physical link that are of interest include:

- **Piconet roles** – whether the local device is a master, or a slave
- **Power** – whether the link is in the correct mode for the traffic the service is consuming or producing

³ This can occur with controllers which do not support scatternet and therefore require specific roles when connecting, or in cases where both remote devices are trying to enforce usage of a certain role (e.g. both devices insist on being master). Both these problems have been observed when performing Bluetooth interoperability testing.

- **Security** – whether the physical link is, for example, authenticated and/or encrypted.

These properties pertain to the physical link, and are important to the application programmer, particularly the power and security aspects. However, a physical link doesn't *directly* carry data itself: instead it supports logical transports that can carry data – logical transports are described in section 4.1.4.

The power consumed by each link depends on the mode in which that link operates. Reducing the time for which the receiver in the controller is active can reduce the power consumption of the link. This is possible because, in the time domain, 'slots' are created between the master and slaves in a physical channel, and it is possible to specify that we only need to listen for traffic in certain slots. This is called 'sniff mode'.

Sniff mode is a very commonly used way to extend battery life. In sniff mode, a periodic arrangement of time slots can be specified as 'anchor' points at which the master and slave commit to a transfer. There is more detail to the arrangement that can allow for a missed transfer at the anchor point, but to summarize, sniff mode reduces the time spent listening on a link for traffic, and therefore the power consumed. The downside can be increased latency and a reduction in bandwidth. Typically a physical link in sniff mode consumes less power than a device requires to listen for device discovery requests and incoming connections. So, for example, a human interface device such as a Bluetooth mouse, or a Bluetooth headset, can connect to the remote device at power on, then place the resulting link into sniff mode. This is more battery efficient than the mouse or headset waiting for an inbound connection.

The default mode of the physical link, which it is in after connection, is 'active' mode. All of the time slots are used to exchange data between the master and slaves in the piconet. In many situations this is wasteful; thus, when the link is required but no traffic is going to flow, the link should be placed into a mode other than active mode. In cases where there will be a lot of traffic flowing, or low latency is required, the link should be placed back into active mode. As an example, a turn-based game such as Battleships or Noughts and Crosses could operate perfectly well in sniff mode. A more interactive game, such as Snake, is likely to require the lower latency of active mode.

Hold mode can be used by Bluetooth hosts when managing more complex Bluetooth networks, particularly when the host wishes to discover and create a physical link to another Bluetooth device. Applications would not typically manipulate this directly, although an attempt by a service to connect to given device may have the side effect of putting another link into hold mode.

Park mode is rarely used but can, in theory, allow more than seven slave devices to be connected to a single master. In addition the master

can broadcast to all of the attached, parked, slaves. However, there are difficulties in making this work in practice, and again is not something a service would be expected to do.

Security can be applied to a link at the service's request. In Symbian OS, the actual link security used is based on the highest security requirements given by any service. There are three aspects to link security – authorization, authentication and encryption. Of these, authorization is independent of the other two, but to use encryption then authentication (i.e., the pairing process) must have taken place at least once between the devices. A more practical discussion on security takes place in section 4.2.7.

4.1.4 Logical Transports

The two most important logical transports are the asynchronous connection oriented (ACL) and the synchronous connection oriented transports (SCO and eSCO). By far the most important to the application programmer is the ACL, as this forms the underlying transport for L2CAP, which is now discussed.

The asynchronous transport

The asynchronous transport (ACL⁴) carries the majority of data in Bluetooth. By default, it is reliable; although it can be configured to be unreliable if required to provide a latency-based isochronous service. There is a choice of packet types available to the ACL transport: the packets can be chosen to optimize the performance of the service in different radio conditions. Generally, due to the dynamic nature of the different radio conditions it usually makes most sense for the Bluetooth hardware to make the decisions on which ACL packets to use.

The ACL transport supports two types of 'logical links' – the ACL-U, and ACL-C; 'U' referring to the user-plane, and 'C' to the control-plane. The ACL-C logical link is used by the lower Bluetooth layers to converse with each other, for example sending messages to disconnect links, change the transmit power and switch encryption on. The ACL-U is the link over which all higher layer protocols in the Bluetooth stack communicate, including, therefore, all services running on Symbian OS.

⁴ A quick word about the name of the transport: 'ACL' was used in 1.1 and earlier versions of the Bluetooth specification to mean 'asynchronous connectionless'. In those versions of the specification there was no documented concept of the physical link (which is connection oriented). Therefore, the ACL appeared to be 'connection oriented' to most readers of the specification. In 1.2 and later versions of the specification, the writers changed 'ACL' to mean 'asynchronous connection-oriented link'; however, they also introduced, correctly, documentation of the physical link. It is the author's view that, given the connection-oriented physical link, it still makes sense to consider the 'ACL' as connectionless.

The synchronous transports

Bluetooth's synchronous transports are one of its key strengths when compared with other wireless technologies: they allow the simple and reliable transportation of voice over a wireless link. Synchronous links are always used with Bluetooth headsets to place and receive phone calls. As the data for these synchronous links does not come through the Symbian OS Bluetooth stack, they are not discussed further here.

4.1.5 L2CAP

The most important protocol in any Bluetooth host is L2CAP – the Logical Link Control and Adaptation Protocol. This is the most common point at which services interface to the Bluetooth stack within Symbian OS.

The name L2CAP implies that it controls and adapts a logical link. The 'logical link' in this case is the ACL-U, discussed earlier. It is useful to abstract away the characteristics of the data buffers in the controller from the application in order to allow applications to be hardware independent. This is the 'adaptation' service of the ACL-U logical link provided by L2CAP.

L2CAP provides a multiplexing, connection-oriented, sequenced-packet datalink service.

Let's quickly run through that – afterwards, if you want more detail on the generic concepts, have a look in Chapter 3. The multiplexing service simply provides many channels over the single underlying ACL-U. The service is connection-oriented as we connect to a specific PSM (see section 4.1.8 for an explanation of what a PSM is) on the remote host before transferring data – this is also useful as it reduces the packet header overhead, as the connection is assigned a channel ID rather than needing to carry all the channel state information in the header in the way that TCP does. And the link is presented in sequenced packet form – we get datagram boundaries that we can use for framing our own application protocol if necessary, and guaranteed, in-order delivery (unless we request otherwise).

The Bluetooth Core specification describes two forms of L2CAP – connection oriented and connectionless. At present there are not many uses for connectionless L2CAP, so it is not discussed here.

Using L2CAP to send isochronous data

Versions of L2CAP prior to version 1.2 of the Bluetooth Core specification are not particularly feature-rich for a multiplexing protocol. For example, there is no per-channel flow control, no per-channel error control, and no segmentation-and-reassembly (SAR). This causes a number of problems – especially with latency-sensitive services such as streaming audio, and in cases where one service using an L2CAP link is receiving data very slowly, and affecting the performance of that link for other services.

As we have seen, by default the ACL transport is ‘reliable’ – in other words, when the hardware is asked to send L2CAP data, it will attempt to send it, possibly forever, until it is acknowledged by the remote Bluetooth hardware. Some services, especially streaming audio-video services do not require such a feature from the Bluetooth hardware, indeed data sent by such services only has a limited lifetime in which it is useful. In these situations an L2CAP entity may instruct the controller to ‘flush’ the data from its sending buffers for a particular ACL transport if it has not successfully sent the data within a specified period of time. This may seem to be a useful feature; however, other services running over the same ACL transport may *expect* a reliable transport (for example, the RFCOMM protocol expects a reliable transport). Without the error control mechanisms of L2CAP 1.2, more complex mechanisms are required to provide optimum transport of AV data and RFCOMM data on the same link. The error-control feature of L2CAP 1.2 provides a much improved way of assuring reliable data transfer alongside time-sensitive data on the same link.

Without per-channel flow control, aggregate flow control at the layer beneath L2CAP is necessary. This implies that some L2CAP channels that have data to transmit are not allowed to do so if there are any other Bluetooth services using the ACL transport that are consuming data slowly, and thus causing the whole link to be flowed off. This problem is solved by the use of per-channel flow control in L2CAP 1.2 as individual L2CAP channels can now perform flow control, rather than it being a property of the whole ACL link.⁵

The SAR feature in L2CAP 1.2 allows the L2CAP multiplexers to provide better latencies to services that require data to be sent within given time bounds. Without SAR, an L2CAP multiplexer is blocked all the time an service’s data packet is being sent by the multiplexer. So, an service that sends low-latency, small packets will have its performance affected by an service that is sending larger packets with no latency requirements since the large packets will make the multiplexer wait longer before sending the small packets. SAR allows the L2CAP entities to split application packets (SDUs) into smaller packets (PDUs), and interleave PDUs from different SDUs. This allows the latency requirements of certain services to be satisfied in a more deterministic manner.

4.1.6 Service Discovery

Service discovery in Bluetooth is performed using the service discovery protocol (SDP) – a rich protocol defined in the Core Bluetooth specification that allows a Bluetooth device to find out about services that are available on another Bluetooth device.

⁵ Remember that the ACL logical link exists between a pair of devices, so this only causes problems if there are multiple services running between the devices. In cases where the remote devices only offer a single service, this problem is far less likely to occur – however, problems can still persist if a service requires more than one L2CAP channel.

Any Bluetooth device that offers services has a service discovery database (SDDB), in which a series of records are stored describing the services that device offers. Services on the device register with the local SDDB, which is then queried by remote devices.

Each service running on a Bluetooth device should advertise a record in the SDDB. An SDP record is a set of attributes, where an attribute is a pair consisting of an ID and arbitrary value. A particularly important type in SDP is the universally unique identifier (UUID). UUIDs are used to identify protocols, profiles, services and service classes in SDP; with the SDP specification making the bold claim that each UUID is guaranteed to be unique across all space and all time.

Structure of the database

The structure of information in the SDDB can be considered in two ways – what we'll call the 'logical' structure and what we'll call the 'physical' structure. The physical structure is basically a serialized version of the data, like blocks on a disc, and is shown in Figure 4.5. This is how the database is described by the Bluetooth specification. There is also a logical view of the data, like a filesystem view of a disc – this is shown in Figure 4.4. You will probably find it much easier to understand the logical

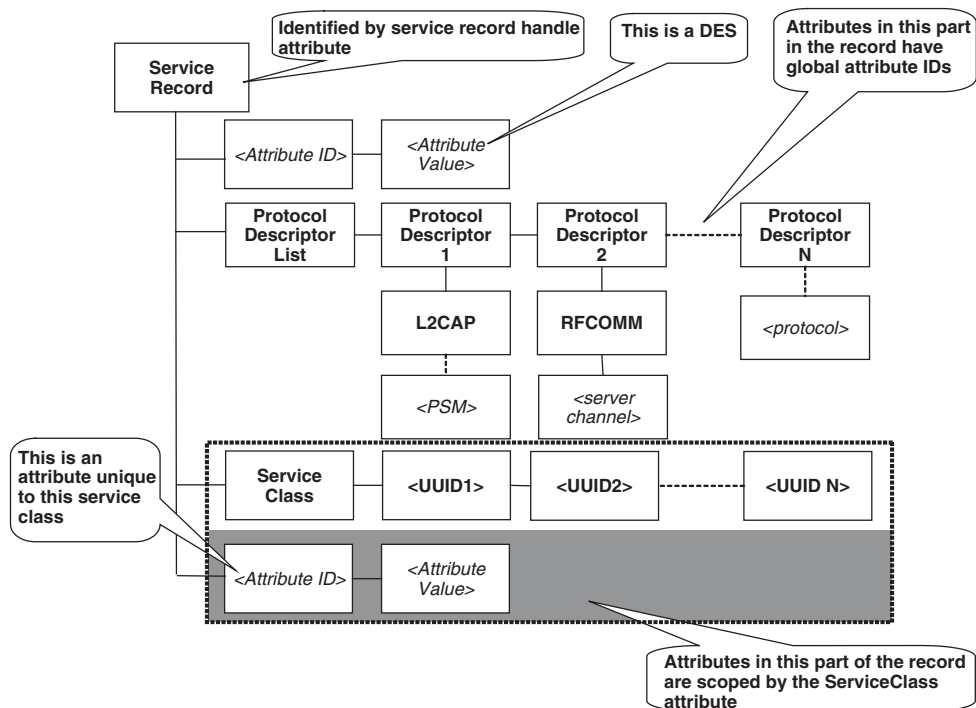


Figure 4.4 A logical view of a service record in service discovery database

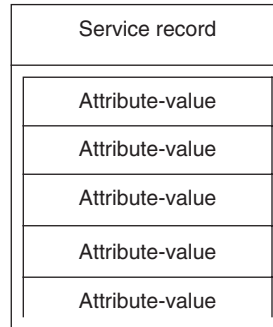


Figure 4.5 A physical view of a service record in the service discovery database

view as it makes certain relationships more obvious. Just bear in mind when using the APIs that the underlying representation uses the physical view – each service record being represented by a list of attribute-value pairs, rather than any form of hierarchical structure being imposed.

In the physical view, the service record handle is just another attribute-value pair. In the logical view, it represents the handle to the entire service record.

Important note: a service that needs to register itself in the SDDB will need a 128-bit UUID to use as a service class. This should be generated once, using a UUID generator, of which there are plenty available on the internet, then hard coded into the application. After this has been done, that UUID should always be used as that application's service class when registering in the SDDB.

Since SDP has a lot of potentially confusing terminology – here's a quick note on our usage. When we say 'attribute' we're referring to the ID and the value. If we mean either the ID or the value we'll say so explicitly.

Attribute ID

An attribute ID is a 16-bit identifier. There are two main groups of attributes IDs – globally defined ones, and ones defined by specific service classes. Globally defined attribute IDs and their usage are defined in the SDP specification, and include the expected things such as *ServiceRecordHandle* and *ServiceClassIDList*, but also *ProtocolDescriptorList* (used for describing the protocol stack required to connect to a service – for example OBEX-RFCOMM-L2CAP, along with parameters used by those protocols – for example RFCOMM server channel number, L2CAP PSM). There are various other global attribute IDs that can be used – some will be seen in the examples later in this chapter. In order

to leave space for more global identifiers to be added, identifiers in the range 0x0000 to 0x01FF are reserved by the Bluetooth SIG.

All other attribute IDs are scoped by a service class – this means that interpreting them requires looking at the service class attribute first. The good news here is that since you’re likely to be using your own service class (unless you’re implementing a standard profile) then you can define any attribute ID in the range 0x0200 to 0xFFFF for your own purposes.

Attribute values⁶

There are two main sorts of value – individual values and lists of values. The lists of values have two subtypes – lists and lists from which one alternative should be selected (these are called *data element sequences* (*DES*) and *data element alternates* (*DEA*), respectively).

So overall there are nine types of value:

- Unsigned integers
- Signed, two-complement integers
- UUIDs
- Text strings
- Booleans
- Data element sequences (i.e., lists where all items are relevant)
- Data element alternates (i.e., a list from which one alternate has to be chosen)
- URLs
- Nil (a NULL type).

Just to make life harder, lists can themselves contain lists, and usually do! With the Symbian OS APIs, which work very much like a SAX-style XML parser, this can make parsing some lists quite tricky as you need to maintain some state yourself indicating which list you’re in.

Searching for services

Now that we’ve covered the basic structure of the database and the types it contains, let’s look at how we query it. Queries are performed using the service discovery protocol (SDP). In terms of protocol layering, it runs directly over L2CAP. It is a request–response protocol that allows access

Service class (implicit scoping)	Attribute ID	Attribute value
-------------------------------------	--------------	-----------------

Figure 4.6 Interpretation of non-global attribute IDs

⁶ Actually, attribute IDs are themselves values of a specific type (16-bit unsigned integers) but we’ll leave that confusing detail aside.

to a remote SDDB. The requests, each of which has a response, are as follows:

- **Service search** – to discover if a specified UUID, or set of UUIDs, exists in any attributes of a service record that is available on a remote device.
- **Service attribute** – to discover values of specified attributes belonging to a particular service record that is available on a remote device.
- **Service search-attribute** – a combination of the above in a single query.

Oddly for a protocol allowing queries of a (possibly large) remote database, there is no ‘abort’ command. Therefore the only way to stop an ongoing request on a remote SDP server is to disconnect the underlying L2CAP connection.

Because SDP responses may contain potentially large amounts of data, a mechanism exists for SDP servers to fragment responses into multiple L2CAP packets. Oddly no equivalent mechanism exists for fragmenting SDP requests.

The oddities of SDP enumerated above (fragmentation, packet types and missing abort) are hidden from applications by Symbian OS. However, due to the complex syntax and structure of SDP attribute values, and the need for applications to be able to access all the possible attribute values that might be returned, the Symbian OS SDP APIs remain complex.

The SDP protocol follows a request/response pattern. Typically the procedure to detect the presence of a service on a remote device is as follows:

1. The SDP client sends a Service Search Request, with a list of UUIDs to search for.
2. The SDP server responds with Service Search Response, which lists the record handles that match the search pattern.
3. If the record handle list is not empty, the client sends an Attribute Search Request with a record handle and list of attributes to retrieve.
4. The server responds with an Attribute Search Response with the value for each attribute requested by the client.

4.1.7 RFCOMM

RFCOMM is a supplementary, but commonly used, protocol in Bluetooth. It provides serial port emulation, including various flow control mechanisms. It cannot provide error control itself, so it relies on L2CAP to provide reliability. As previously discussed, it is preferable that L2CAP uses its own error control mechanism to provide reliability, rather than

requiring the ACL to stay in reliable mode, as this allows RFCOMM to coexist with time-sensitive data on the same link.

RFCOMM is based on the GSM 07.10 protocol, which in turn is based on the HDLC protocol. HDLC provides the underlying frames that are used during RFCOMM connection setup, teardown and data exchange. GSM 07.10 adds an 'internal' protocol to allow the exchange of control messages between RFCOMM multiplexers to configure channels attached to the multiplexer, send test commands, apply flow control etc. Note that, although the 07.10 protocol allows for the non-multiplexed exchange of data (the so-called 'disconnected mode'), this is not supported in RFCOMM.

Since the protocol provides a simple datalink, there are not a great deal of user services other than the provision of a byte pipe. However, there are some features available that merit discussion due to RFCOMM's ability to emulate a serial connection.

The flow control mechanisms in RFCOMM are:

5. Per-channel stop/go flow control between RFCOMM entities.
6. Multiplexer stop/go flow control between RFCOMM entities.
7. Per-channel credit-based flow control between RFCOMM entities.
8. RTS/CTS emulated RS-232 flow control between applications using RFCOMM.
9. DTR/DSR emulated RS-232 flow control between applications using RFCOMM.
10. XON/XOFF emulated RS-232 flow control between applications using RFCOMM.

Notice that the first three of the flow control mechanisms are applied between RFCOMM entities, not the applications using RFCOMM. Thus the application cannot directly stop or start the data flow. The final three are effectively ways for applications to believe they are using an RS-232 cable using standard flow control mechanisms. The RFCOMM entities do not honour the flow control signals in these cases – they merely convey them to the peer applications. Since a wireless technology has a much higher latency than RS-232 it is of questionable benefit for an application to rely on the emulated flow control to actually perform flow control. Instead, it is far better for the application to delegate flow control to those built into RFCOMM (mechanisms 1–3 above).

Other emulated RS232 services provided by RFCOMM include the ability to configure emulated port settings such as baud rate, number of data and stop bits, parity etc. Again, RFCOMM does not honour these settings – so, for example, it does not throttle the data sent between RFCOMM entities to match the emulated baud rate. The settings are

merely conveyed so that applications can believe they are running over an RS-232 link.

4.1.8 Ports

Both L2CAP and RFCOMM are connection-oriented multiplexing protocols; thus each requires the service to consider on which port it wishes to receive connections over which to transport the service's data.

There are subtle semantic differences between L2CAP and RFCOMM ports that should be noted.

L2CAP ports

L2CAP ports are called 'protocol service multiplexers', or PSMs. They allow multiple channels *from the same device* to be connected to the same PSM; each time an L2CAP connection is created from a device to a remote PSM, the remote L2CAP entity assigns a unique channel identifier (independent from the PSM) to the connection (this identifier is hidden from the application, but the `RSocket` forms a handle to it). There is no concept of a 'source' PSM, unlike TCP and UDP, which have the concept of source ports.

Different multiplexers handle connections to or from other remote devices (there is one multiplexer in the Symbian OS L2CAP implementation per remote device) so there is no risk of collision between channel identifiers from different devices. This is also transparent to the application, and does not change the way PSMs are handled.

The range of L2CAP port numbers is split into two parts by the L2CAP specification: those below 0x1001, which are reserved by the Bluetooth SIG, and those greater than or equal to 0x1001 which are available to users. Another rule regarding PSM values is that the final octet must have the least-significant bit (lsb) set. The L2CAP specification actually allows PSMs to be extended beyond 16 bits (by clearing the lsb of the second octet) – however Symbian OS, as with many other implementations, does not support this. Of course this still leaves around 30,000 ports available!

It is important to note that L2CAP ports in the user range are *never* 'well known', unlike many services using TCP and UDP which typically publish their port. The reason for this is that SDP allows the port value to be retrieved dynamically. The corollary of this is that an application running over L2CAP should not register using a hard-coded port. To aid dynamic port selection, there is a mechanism for the application to request a free port from the local L2CAP entity. This is an important feature for running applications both over a protocol that does not use well-known values *and* on an open platform such as Symbian OS, where many such applications may be present.

Note that if your service requires more than one channel – for example to carry two different traffic types – then you could connect them to

the same PSM. This is how the Bluetooth A2DP profile operates – it uses multiple connections to the same port, with the sequence of the connections indicating the purpose of that channel. In this case SDP does not provide a mechanism to indicate either the purpose of the different connections, or the quantity of connections required – these rules are statically held by the service implementations. In most circumstances though it is preferable to use multiple ports, because using the order in which the connections are made to determine the purpose of the channel can add a great deal of complexity. However, as is the case with A2DP, there are occasionally some situations in which it is useful to multiplex onto the same port (the result being that Bluetooth baseband packets can be filled more completely).

RFCOMM ports

RFCOMM ports are called ‘server channels’. Unlike L2CAP PSMs, they do *not* allow multiple channels on the same port between two given devices. This is because there is no dynamically assigned channel identifier that is independent from the server channel (actually, there *is* a channel identifier that can be a different value to the server channel, but it is not *independent* from the server channel).

Like L2CAP, there is an RFCOMM multiplexer per remote device in Symbian OS. Therefore as channels are identified by a (source address, destination address, server port) tuple, there is no problem with *different* remote devices connecting to the same server channel.

As with L2CAP, SDP is always used to retrieve port numbers for a given service – therefore there are *no* well-known ports in RFCOMM. Again, as with L2CAP, the RFCOMM protocol in Symbian OS provides a mechanism for the application to obtain a free server channel.

Unlike L2CAP, RFCOMM has far fewer ports: indeed only around 30; with no reserved/user split. A number of the RFCOMM server channels will already be in use after the Symbian device has booted; these will be listening for connections from Bluetooth headsets, for receiving objects using the OBEX protocol, or for receiving connections to enable the device to act as a Bluetooth modem.

The RFCOMM specification does allow an increase in the number of ports available to a local device, however, RFCOMM is considered a somewhat ‘legacy’ protocol now that L2CAP is gaining more features that allow any form of service to be developed and therefore Symbian OS does not implement this feature.

When to Use L2CAP or RFCOMM

Since L2CAP and RFCOMM are both datalink protocols, there is a decision to be made as to which should be used when implementing a Bluetooth service.

The advantage of RFCOMM is that it is guaranteed to provide a flow control service individually to each service, unlike L2CAP which can only do this when both ends of the link support L2CAP 1.2 or greater. It also provides an RS-232 emulation service, but the need for that in newly developed wireless services is arguably non-existent. RFCOMM is also guaranteed to provide a reliable service to its clients – however, it does this in a way that can impact other services running on the same device. A final advantage is that for some services the stream interface of RFCOMM may be useful, although it is trivially easy to simulate a stream protocol using the L2CAP sequenced packet service.

The advantage of L2CAP is that it is, by definition, a lighter-weight protocol because it has removed the extra connection setup, and frame headers and trailers that are yielded through the use of RFCOMM (which is, of course, running over L2CAP).

If you can be sure that the two devices running your service will be using Symbian OS v8.1 and later, then you can rely on the availability of error and flow control services at the L2CAP layer – so there is no need to use RFCOMM to obtain these services. Furthermore, the SAR service within L2CAP 1.2 will help to maintain better protocol performance of many independent Bluetooth services. L2CAP can make certain higher layer protocols easier to implement through its datagram semantics. Of course, the service can provide a shim over L2CAP sockets to make them appear as a stream interface if it so desires. Finally, L2CAP has a greater number of ports, and possesses more sophisticated semantics for the support of multiple connections.

It should be noted that a service may choose to use L2CAP *and* RFCOMM at the same time: it might make sense to have a ‘control’ channel using L2CAP to exchange the service’s control messages, but to use RFCOMM – to the same service on the same remote device – to send bulk data. SDP can store this information in the service record as a `ProtocolDescriptorList` (say for the L2CAP connection), and an `AdditionalProtocolDescriptorList` (for the RFCOMM connection).

4.2 Bluetooth in Symbian OS

4.2.1 Overview in Symbian OS

Symbian develops its own implementation of the Bluetooth host. This greatly increases Bluetooth interoperability between any Symbian OS-based devices. It also increases the functional compatibility between the applications and the underlying Bluetooth subsystem in the OS. The Symbian OS host is designed to run over any controller. Note that different controllers have different feature sets; thus some features discussed within this chapter may not be available on all platforms. However, any unsupported features will fail gracefully at the API level.

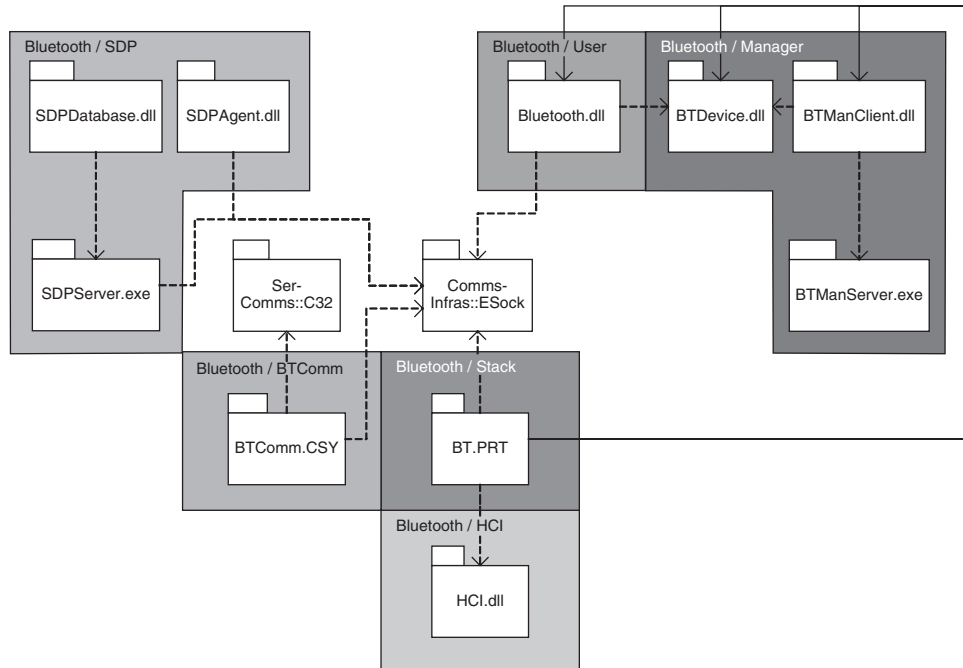


Figure 4.7 The Bluetooth subsystem in Symbian OS

The Symbian OS Bluetooth host consists of a number of protocols described within the Bluetooth specifications: these are L2CAP, SDP, RFCOMM, OBEX, AVCTP, AVDTP and BNEP. The profiles within Symbian OS (as delivered to licensees of Symbian OS) are GAP, SPP, AVRCP, GAVDP and PAN. These will be discussed in this chapter.

A large part of the Bluetooth subsystem is available via ESOCK APIs – details on the generic APIs are available in Chapter 3, but here we show the Bluetooth-specific details.

4.2.2 Discovering and Connecting to Devices

Discoverability of the Local Device

Applications within Symbian OS cannot set the discoverability of the local device. The reason for this is that it is a user-controllable, device-wide property that the user expects to set from a single location. If applications interfere with the discoverability of a device then the user can feel ‘out of control’. Symbian OS supports the limited discoverable mode; however, there are, at the time of writing, no UIs that take advantage of this feature: the UIs only present the general- and non-discoverable modes.

The Bluetooth stack in Symbian OS actually has a more complex usage of the discoverability setting than the binary indication of most UIs implies: the Bluetooth device is never made discoverable unless there are Bluetooth services running to which connections can be made. In

practice the SDP server is always listening for connections, so whenever the user requests the device be made visible then it occurs. The user can, of course, override this to become invisible – in this case they can limit the devices that connect to those that they have connected with previously.⁷

Device discovery using basic symbian OS mechanisms

There are two options for discovering other Bluetooth devices, depending on whether you need user interaction or not. First we'll cover the underlying device discovery system which can be used directly if required, and on which the device selection UIs are based, then we'll go on to look at the specific UIs provided by S60 and UIQ for performing this task.

Bluetooth devices can be found using `RHostResolver`, along with `TInquirySockAddr`.

Each protocol in Symbian OS can expose a 'host resolver': the semantics and functions of the host resolvers depends on the protocol exposing them. Only the 'BTLinkManager'⁸ protocol in the Symbian OS implementation of Bluetooth can create host resolvers.

The `TInquirySockAddr` allows configuration by an application of the operations presented by `RHostResolver`. The `TInquirySockAddr` class is also used to retrieve the results from the Bluetooth device discovery.

The important methods on `TInquirySockAddr` are:

```
IMPORT_C TInquirySockAddr();
IMPORT_C TBtDevAddr BtAddr() const;
IMPORT_C void SetBtAddr(const TBtDevAddr& aRemote);
IMPORT_C static TInquirySockAddr& Cast(const TSocketAddr& aSocketAddr);
IMPORT_C TUint16 MajorServiceClass() const;
IMPORT_C TUint8 MajorClassOfDevice() const;
IMPORT_C TUint8 MinorClassOfDevice() const;
IMPORT_C void SetIAC(const TUint aIAC);
```

Notice that one of the methods allows the setting of the IAC, the inquiry access code. Two values are specified: `KGIAC` and `KLIAC`. If the latter is used then a limited discovery will be performed; specifying the former will yield a general discovery. Since most user interfaces on most devices do not support being in limited discovery mode, it is rarely worth performing a limited discovery.

⁷ Even in non-discoverable mode it is possible for someone to 'brute-force' a discovery by attempting to connect to every single device that could possibly exist. Because of this, non-discoverable mode should not be considered a strong security measure, but rather a way of making the device harder to find without a concerted effort.

⁸ This protocol is not defined in the Bluetooth specification. Symbian OS implements a layer separate to, and below, L2CAP that manages and models the logical transports and physical links being used. This ensures the implementation of L2CAP follows the L2CAP specification much more closely.

Example 4.1 shows how the two classes are used to begin a device discovery, and retrieve the results:

```
class CBtDeviceDiscoverer : public CActive
{
public:
    ...
    void DiscoverDevicesL();
    virtual void RunL();
    ...
private:
    RSocketServ iSocketServ;
    RHostResolver iHostResolver;
    TInquirySockAddr iInquiryAddress;
    TNameEntry iNameEntry;
};

void CBtDeviceDiscoverer::DiscoverDevicesL()
{
    TProtocolDesc pInfo;
    _LIT(KLinkMan, "BTLinkManager");
    TProtocolName name(KLinkMan);
    // precondition: iSocketServer has been Connect()ed successfully
    User::LeaveIfError(iSocketServer.FindProtocol(name, pInfo));
    // Open an appropriate host resolver
    User::LeaveIfError(iResolver.Open(iSocketServer, pInfo.iAddrFamily,
                                     pInfo.iProtocol));

    // Set up inquiry address
    iInquiryAddress.SetIAC(KGIAC);
    iInquiryAddress.SetAction(KHostResInquiry);
    iResolver.GetByAddress(iInquiryAddress, iNameEntry, iStatus);
    SetActive();
}

void CBtDeviceDiscoverer::RunL()
{
    {
        if(iStatus.Int() == KErrNone)
        {
            //We found another device, fetch the address
            TBTDevAddr bdaddr = TBTSockAddr::Cast(iNameEntry().iAddr).BTAddr();
            // Do something with address here (eg. add to an array of discovered
            // devices
            // Now get the next device from those found
            iResolver.Next(iNameEntry, iStatus);
            SetActive();
        }
        else if(iStatus.Int() == KErrHostResNoMoreResults)
        {
            // the search has finished
        }
    }
}
```

Example 4.1 Discovering devices without using a UI

Although devices may be found using RHostResolver (mapping onto the device discovery procedure in the Generic Access Profile), there is no guarantee that the device provides the service sought. For example,

a device such as a phone is unlikely to provide a printing service. SDP allows devices to be queried to discover what services they provide. SDP also allows the discovery of features within the services provided, settings used by a service, and the parameters required to connect to a service.

The Bluetooth stack also places devices in the host resolver at times other than during the inquiry phase. This allows, for example, the class of device of a remote device that has never been found in a device discovery and that connects to the local device to be stored by the stack. The application, having received a successful connection at the L2CAP or RFCOMM level can then query the host resolver for details of a single device (for example, a device which has just connected to that application).

To query the host resolver for details of a specific device the following code can be used – in this case we’re connected to a remote device and want to know if it supports any object transfer services:

```
// iSocket is connected to a remote device
TBTSockAddr socketAddress;
(void)iSocket.RemName(socketAddress);
//hostResolver is a successfully opened Host Resolver
TInquirySockAddr inqAddr(socketAddress);
inqAddr.SetAction(KHostResCache);
TNameEntry name;
hostResolver.GetByAddress(socketAddress, name); // note we use the
        synchronous version here as we are looking something up from the local
        device cache

// Extract CoD information
TUInt16 majorServiceClass = inqAddr.MajorServiceClass();

// Now see if the device supports the object transfer major service class
if (majorServiceClass & EMajorServiceObjectTransfer)
{
    // we should now perform an SDP query to see if it supports the type of
    // object transfer (eg. OPP) that we require
}
```

Example 4.2 Checking CoD information for a connected device

Note that we could have used much of the second half of this example to perform the same functionality on the results of a device discovery to filter out devices that did not support the type of functionality we were interested in.

Device discovery using standard UI mechanisms

In order to perform device discoveries where user interaction is required it is best to use the device discovery mechanism provided by the UI platform – in the case of S60 this is using a notifier through the RNotifier interface, and in the case of UIQ it is CQBTUISelectDialog.

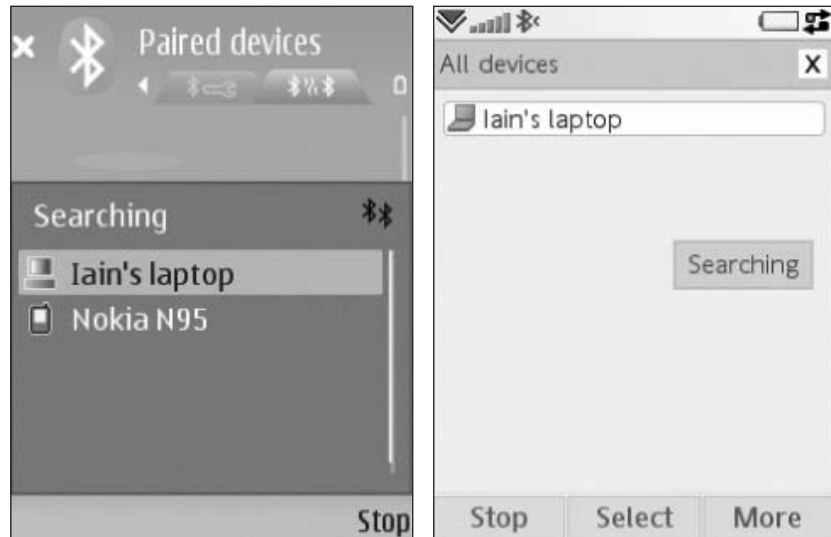


Figure 4.8 S60 and UIQ device selection dialogs

In the S60 case, the implementation of device selection through the notifier framework means that you can use the device selection routines from any thread, whether it has an UI environment or not. Here's a brief example of its use:

```
class CS60BtDeviceDiscoverer : public CActive
{
public:
    ...
    void DiscoverDevicesL();
    virtual void RunL();
    ...
private:
    RNotifier iDevDiscovNotif;
    TBTDeviceSelectionParamsPckg iSelectParamsBuf;
    TBTDeviceResponseParamsPckg iResponseParamsBuf;
    TBTSockAddr iAddr;
};

void CS60BtDeviceDiscoverer::DiscoverDevicesL()
{
    User::LeaveIfError(iDevDiscovNotif.Connect());
    iDevDiscovNotif.StartNotifierAndGetResponse(iStatus,
        KDeviceSelectionNotifierUid, iSelectParamsBuf, iResponseParamsBuf);
    SetActive();
}

void CS60BtDeviceDiscoverer::RunL()
{
    if(iStatus.Int() == KErrNone)
    {
```

```

// iResponseParamsBuf now contains details of the device the user
// selected
iSockAddr.SetBTAddr(iResponseParamsBuf().BdAddr());
// once other params (eg. security, port) set on iAddr it can be used
// to
// connect a socket
}
else // an error occurred
{
    User::Leave(iStatus.Int());
}
}

```

Example 4.3 Discovering devices using the S60 device discovery UI

To tailor the search to a more specific subset of devices (e.g. devices offering some form of rendering capability) see the methods on `TBTDeviceSelectionParams`.

Connectability

As with discoverability, Symbian OS applications cannot control the connectability of the local device. Certain UIs will have a widget that the user can control that *may* directly control the connectability – though the ‘connectable’ tag is much less meaningful to a user than ‘discoverable’. Often the UI widget that may affect connectability refers to turning Bluetooth on or off; depending on the combination of UI and hardware this may range from toggling connectability to toggling the actual power status of the Bluetooth hardware.

Also, similarly to discoverability, the Bluetooth stack in Symbian OS will not make itself connectable if there are no services to connect to. Furthermore, Symbian OS does not allow the device to be in a state where it is discoverable, but non-connectable.

Pairability

The Bluetooth stack in Symbian OS and the UI interact during the Bluetooth pairing process to obtain the passkey from the user. Once the user has entered a passkey it is returned to, and stored by, the Bluetooth subsystem. Today, all UI platforms leave the device in pairable mode at all times. There is no API to directly control whether the device is globally pairable or not.

Class of device

The class of device value is managed by two mechanisms in Symbian OS: the device class is fixed by the device manufacturer since it is a static value declaring the perceived appearance of the device to the end user. So for example a Symbian device may either be a ‘smartphone’, a regular

‘mobile phone’, or perhaps a ‘laptop’. The device manufacturer sets this value at device creation time.

Applications can register the service class value for their corresponding use case (if applicable – not all services will need to update the service class field).

The applications make the updates to the service class by either of two mechanisms:

- Publish and Subscribe (P&S)
- Using a `SetOpt()` on their socket.

Clearly the latter requires the service has access to the underlying Bluetooth socket. Despite this limitation, if the socket is available it is the preferred method to set the service class, due to two problems using the P&S method – the relatively high capabilities required to perform the operation and the possibility of race conditions.

The P&S mechanism is useful for services that, for example, run over OBEX (see Chapter 10 for more details on running services over OBEX), as the client does not have access to the underlying socket to use the `SetOpt()` call to set the service class. There are a number of Bluetooth applications which may elect to run a service over OBEX; in these cases, those applications will need to use the P&S mechanism to set the relevant service class bits – specifically the ‘object transfer’ bit. However, there is a problem with this approach – in order to write to this P&S key, applications require both `LocalServices` and `NetworkControl` capabilities. This essentially makes it impossible for most applications to perform this operation.

Note that from Symbian OS v9.2 onwards, the OBEX stack will automatically set the object transfer bit in the service class when listening on a Bluetooth transport. It does this using the socket `SetOpt()` call, which does not require the `NetworkControl` capability.

In v9.1 there is no direct solution to this issue – although usefully the object transfer bit will be set whenever the default OBEX service is running on a device. Currently all UI platforms keep this service running whenever devices are connectable, so in practice this should not turn out to be a problem.

The other issue with using the P&S mechanism is the possibility of race conditions. These can occur when two clients need to enable a new service class bit – the sequence for a client should be read, modify, write. There is a small possibility that two clients will interleave their updates, such that the first one reads the current value, then the second one does, then the first one writes the new value, then the second one

writes its new value. In that case the update from the first client will be lost. Reading back the service class after setting it to ensure the expected update occurred may appear to be a workaround for this, however, there is no guaranteed solution to the problem as scheduling issues can cause the read to return the expected value. The optimum solution to this problem is always to use the `SetOpt()` in preference to the P&S key.

A quick note on filtering using the class of device. Neither the major or minor device classes should be used for filtering – the intent of these is to present, for example, an informative icon to the user of the device performing the device discovery. Major service class can be used to filter devices *out* of a search if they do not claim to provide the appropriate high-level service class, but any devices that do have the appropriate bit set still need to be queried using SDP to ensure they provide the particular service that the application is searching for. For example, we can safely assume that devices without the object transfer bit set do not offer the FTP (OBEX file transfer profile – a standard way of transferring objects between devices) service, but any that do have it set may not offer the FTP service either – they may well be offering another, unrelated, object transfer service. Thus once we have identified a candidate set of devices offering the service, we should then perform SDP queries for that service on the device before attempting to connect to it.⁹

When using the P&S method, `TBTDeviceClass` is used when dealing with device classes. It has getters for each part of the device class: major device, minor device and major service classes. There are no setting methods – setting is only allowed at construction time.

Here's an example in which we add in the networking major service class using the P&S key:

```
TInt currentCodAsTInt;
User::LeaveIfError(RProperty::Get(KPropertyUidBluetoothCategory,
    KPropertyKeyBluetoothGetDeviceClass, currentCodAsTInt));

TBTDeviceClass currentCod(currentCodAsTInt);
TUint16 newServiceClass =
    static_cast<TUint16>(currentCod.MajorServiceClass() |
        EMajorServiceNetworking);

TBTDeviceClass newCod(newServiceClass, currentCod.MajorDeviceClass(),
    currentCod.MinorDeviceClass());

User::LeaveIfError(RProperty.Set(KPropertyUidBluetoothCategory,
    KPropertyKeyBluetoothSetDeviceClass, newCod.DeviceClass()));
```

Example 4.4 Adding a new bit into the major service class using P & S

⁹ In this case we must perform the SDP search anyway to discover on which RFCOMM server channel the default OBEX service is listening. But even in cases where there is no dynamic port information registered in SDP (PAN profile, for example) we still need to perform the query to ensure the precise service we require is available.

There is much less effort involved in using `SetOpt()`, we simply ask for the bit we want added to the major service class:

```
// iListeningSocket is a constructed CBluetoothSocket (of any protocol
//                                     supported by CBluetoothSocket)
TInt error = iListeningSocket.SetOpt(KBTRegisterCodService, KSolBtSAPBase,
                                     EMajorServiceNetworking);
```

Example 4.5 Adding a new bit into the major service class using a `SetOpt()`

Notice that the level of the option is `KSolBtSAPBase` – this implies that it can be sent to any Bluetooth socket.

4.2.3 Logical Transports

In Symbian OS, although there are classes published pertaining to use of the synchronous links, there is very little need to use them. This is because the use of synchronous links is currently only specified by the Hands Free Profile, which is already implemented in shipping Symbian OS-based devices. Further, the Bluetooth stack in Symbian OS does not provide any real-time guarantees regarding the transport of data via a synchronous link. And finally, Bluetooth controllers are often configured to require data that is to be transported over a synchronous link be given to the Bluetooth hardware via a separate physical interface to non-synchronous data – the management of this routing does not have a published API. As a result, we will not be covering the published classes relating to synchronous links, as they offer nothing of general use.

At present there are no APIs that refer to the ACL logical transport *per se*; however, to influence the choice of ACL packets that the physical link may carry, see section 4.2.4.

4.2.4 Physical Link and Physical Channel

While there is no particular need to understand the physical channel concept when implementing applications in Symbian OS, it is highly recommended for applications using Bluetooth to consider the physical link over which they are running.

Physical link control

One of the key features of Bluetooth as a technology is that it is designed with battery efficiency in mind; far more so that, say, the 802.11 family of standards (aka wireless LAN).

Each Bluetooth service has a responsibility to use the physical link in a manner that considers battery life. We'll briefly recap the modes of the physical link that we discussed in section 4.1.3.

Hold mode is used internally by the Symbian Bluetooth stack to free up capacity when discovering new devices. Hold mode is not meaningful to individual applications – therefore there is no API for applications to place a link into hold mode.

Since Symbian OS v8.0, the Bluetooth stack does not automatically transition links from active to sniff mode or vice versa. Instead it requires information from applications about their requirements for the power mode of the physical link as the stack itself cannot know the nature of the traffic flow on that physical link. These requests are then arbitrated to determine the actual mode of the physical link. Applications should be aware that data can be sent over a link which is in sniff mode, but that the latency experienced will be higher than for a link in active mode. However, as discussed earlier, there is a cost in terms of battery life to using active mode – therefore it is advantageous to transition links to sniff mode whenever possible.

Thus well-written Bluetooth applications should contain some logic to indicate their preference for the power mode of the link. If an application wishes to exchange a ‘large’ amount of data quickly, then it should request that the Bluetooth stack prevents the link going into sniff mode. If the application gets to a state where it has finished its high volume data transfer then it can cancel the request to the Bluetooth stack for active mode, and may optionally indicate that sniff mode is now its preferred option.

Of course, some applications may not wish to contain this level of Bluetooth-specific logic. In this case the physical link may transition into any state, including sniff mode. The application may notice an increase in latency, or a reduction in bandwidth, of the link when this occurs.

Various properties of the physical link can be adjusted using `RBTPhysicalLinkAdapter`. This allows an application to request various modifications to the underlying Bluetooth physical link to improve battery life, performance, etc. Here’s the API:

```
IMPORT_C TInt PreventRoleSwitch();
IMPORT_C TInt AllowRoleSwitch();
IMPORT_C TInt RequestMasterRole();
IMPORT_C TInt RequestSlaveRole();
IMPORT_C TInt PreventLowPowerModes(TUint32 aLowPowerModes);
IMPORT_C TInt AllowLowPowerModes(TUint32 aLowPowerModes);
IMPORT_C TInt ActivateSniffRequester();
IMPORT_C TInt ActivateParkRequester();
IMPORT_C TInt CancelLowPowerModeRequester();
IMPORT_C TInt RequestChangeSupportedPacketTypes (TUint16 aPacketTypes);
IMPORT_C void NotifyNextBasebandChangeEvent (TBTBasebandEvent&
    aEventNotification, TRequestStatus& aStatus, TUint32 aEventMask =
    ENotifyAnyPhysicalLinkState);
IMPORT_C void CancelNextBasebandChangeEventNotifier();
```

As discussed, the various calls are ‘request’-based, rather than mandatory – all current requests are arbitrated by the Bluetooth stack, with

the ‘overall’ mode being determined by considering the best possible mode for the link given the requests made. Note that a request is considered current until the corresponding deactivation is received (e.g. `PreventRoleSwitch()` is cancelled by `AllowRoleSwitch()`); the appropriate cancel method is called (e.g. `ActivateSniffRequestor()` is cancelled by `CancelLowPowerModeRequestor()`); or the request is superseded by more recent request (e.g. `PreventLowPowerModes(EAnyLowPowerMode)` is replaced by `PreventLowPowerModes(ESniffMode)`).

The `RBTPhysicalLinkAdapter` provides a link-state observation method, `NotifyNextBasebandChangeEvent()`. This allows clients to observe the state of the physical link, although it is not recommended that applications try and take any action as a result of this – applications should make their requests for link configuration in all cases, rather than trying to monitor the link state and alter it if necessary. Apart from anything else, this makes the application coding much easier!

In particular, we recommend that applications do not try to ‘police’ the state of the physical link – for example, an application should not fail to provide a service if the local device does not become the master, or the link cannot transition to active mode. This is because the link state cannot be guaranteed – other applications may well be running over the same physical link, or the remote device may disallow any change to the link. Attempts to enforce link state often end in interoperability problems, as there is no one point that has all the information required to configure all the possible links in use between interconnected devices. Therefore the best way to maximize the chances of successful interoperability is to leave the final decision on link state to the lower levels of the software and hardware.

4.2.5 L2CAP

Symbian OS presently implements only the (far more common) connection-oriented L2CAP. Despite it being connection-oriented, it is a sequenced packet protocol – which means it works slightly differently to the connection-oriented protocol most people are familiar with – TCP.

As discussed in section 4.1.5, version 1.2 of the Bluetooth Core Specification added flow control, error control and segmentation and reassembly (SAR) features to L2CAP. All of these features are implemented in Symbian OS v8.1 and later. However, to enable their use, the remote Bluetooth device must also have implemented this improved version of L2CAP (thus Bluetooth connections between recent Symbian OS devices are likely to use the improved L2CAP). Each of the features added in this version of L2CAP improves the performance of L2CAP when running more than one protocol or profile to a remote device.

L2CAP is accessed via a socket API – either `RSocket` or `CBluetoothSocket` (`CBluetoothSocket` is covered in section 4.2.7).

L2CAP Options

An advantage of using L2CAP instead of RFCOMM is that the configuration of the channel is far more flexible than with RFCOMM.

L2CAP sockets are datagram-based; the client therefore has to be aware of the L2CAP packet sizes in use on the link, especially as they can be different in each direction. Whilst exchanging data, the application has to ensure that the descriptors they send and receive will fit into the maximum transmission unit (MTU) that L2CAP has negotiated.

L2CAP specifies a default MTU of 672 bytes in each direction. Note this default is a 'default default' – some Bluetooth services may expect a different, default MTU which must be supported (for example, the Bluetooth PAN profile requires that all implementers negotiate an MTU of *at least* 1691 bytes). Finally, the L2CAP specification specifies a minimum MTU that all Bluetooth devices are required to support – 48 bytes.¹⁰

Although there are various specified defaults, the application *must* make the call to check the actual negotiated MTU – at least in the sending direction. If a client sends data that is too large to fit inside an L2CAP packet, the send operation will be completed with `KErrTooBig`. The negotiated MTUs for the link are available once the L2CAP connection to the remote device, and hence the `Connect()` call on the `RSocket` API, has completed.

The class with which to configure L2CAP channels is `TL2CapConfig`. `TL2CapConfigPckg` is a useful typedef that uses the `TPckgBuf` template class in Symbian OS to assist in the transfer of a `TL2CapConfig` instance to and from a socket. Both are published in `bttypes.h`.

The SDU the application intends to use for transmission is configured using `SetMaxTransmitUnitSize()`, and for the receive path, `SetMaxReceiveUnitSize()`.

The reliability of L2CAP channels can be configured: the application can request an L2CAP channel be unreliable or reliable using `ConfigureReliableChannel()` and `ConfigureUnreliableChannel()`. Of course, 'unreliable' in this case means that data will be dropped when it has been waiting for transmission longer than the specified time, rather than any deliberate attempt to make the channel unreliable!

An application can also influence the scheduling of L2CAP using `ConfigureChannelPriority()`. This allows an application that may

¹⁰ L2CAP (in all versions of the specification) provides a service to higher layer protocols and applications which effectively allows the application layer to notify its peer about *its* packet sizes. This is the MTU of the L2CAP 'SDUs' (the application's packets). In Bluetooth 1.1 this MTU *is also* the size of L2CAP packets ('PDUs'). Although the SDU MTU notification service is retained in Bluetooth 1.2 and later, the SDU size is *not necessarily* the L2CAP PDU size (due to the SAR service). In these later versions of L2CAP the PDU size L2CAP negotiates is hidden from the application, although the application must still check the negotiated MTU even in these cases, as the remote *application*, rather than L2CAP implementation, will also have restrictions on the maximum packet size it can handle.

have two sockets – one for ‘commands’ and one for ‘data’ – to indicate to L2CAP that the ‘command’ socket should be scheduled with a higher priority than the ‘data’ socket.

When configuring a reliable channel the application must specify a time for which L2CAP will retry sending data packets. If this time expires L2CAP will disconnect the logical channel.

When configuring an unreliable channel the application must specify a timeout for the data – when data is submitted to L2CAP for delivery over an unreliable channel, L2CAP starts a timer for each application SDU. When the SDU age reaches the specified value, it is dropped by L2CAP.

Each of the time values just discussed is in milliseconds.

```
IMPORT_C TL2CapConfig();
IMPORT_C TInt SetMaxTransmitUnitSize(TUint16 aSize = 0xffff);
IMPORT_C TInt SetMaxReceiveUnitSize(TUint16 aSize = 0xffff);
IMPORT_C TInt ConfigureReliableChannel(TUint16 aRetransmissionTimer);
IMPORT_C TInt ConfigureUnreliableChannel(TUint16 aObsolescenceTimer);
IMPORT_C TInt ConfigureChannelPriority(TChannelPriority aPriority);
```

The following snippet shows how to configure an unreliable channel with a maximum transmit SDU size of 1500 bytes, a maximum receive SDU size of 100 bytes and schedule as high priority:

```
// iL2CapConfig is of type TL2CapConfigPckg
iL2CapConfig().SetChannelPriority(TL2CapConfig::EHigh);
iL2CapConfig().SetMaxReceiveUnitSize(100);
iL2CapConfig().SetMaxTransmitUnitSize(1500);
iL2CapConfig().ConfigureUnreliableChannel(200); // milliseconds before
data expires
result = iSocket.SetOpt(KL2CAPUpdateChannelConfig, KSolBtL2CAP,
iL2CapConfig);
```

Example 4.6 Configuring L2CAP to provide an unreliable channel

As mentioned earlier, for any L2CAP connection, the application must retrieve the actual negotiated SDU MTU:

```
TInt mtu = 0;
TInt err = iSocket.GetOpt(KL2CAPGetOutboundMTU, KSolBtL2CAP, mtu);
```

Example 4.7 Retrieving the negotiated outbound MTU after an L2CAP socket has been connected

Other L2CAP services

One other L2CAP service to mention is the echo request. Similarly to the echo-request type in ICMP, this service allows an application to test whether the L2CAP entity at the remote device is ‘alive’, and allows the application to provide data that will be sent to the remote device. This

is of limited use to applications because the L2CAP specification does not provide guarantees about the delivery or return of any data supplied in the echo packet. Furthermore, the Symbian OS implementation does not indicate to a receiving application that an echo has been received or replied to – the Bluetooth stack silently replies to the incoming echo. Therefore it is not suitable for application-level keep-alive services.

4.2.6 RFCOMM

In Symbian OS RFCOMM can be accessed through three alternate APIs – `RSocket`, `RComm` and `CBluetoothSocket`. The `RComm` class presents a serial port abstraction to an application; therefore any application using the `RComm` interface can use a virtual Bluetooth serial port.

Ironically, if emulated serial port signals are required, it is actually better to use the `RSocket` class rather than `RComm`. This is because the Bluetooth implementation of `RComm` does not map the `RComm` calls relating to many of the RS-232 lines onto RFCOMM. In contrast, a client of the `RSocket` can make `RSocket::Ioctl()` calls to send and receive emulated RS-232 signals. Thus the `RComm` interface to RFCOMM is not recommended for any new code.

Additionally, as mentioned in section 4.1.7, the use of emulated RS-232 signals for application-level flow control does not work particularly well over the asynchronous Bluetooth link, due to the higher latencies and data rates involved. One of its major limitations is that it does not support, in the standard Symbian OS release, DCE mode – so incoming connections are not supported on the `RComm` interface. Therefore we will concentrate on using RFCOMM through the `RSocket` or `CBluetoothSocket` interfaces.

RFCOMM sockets have different semantics to L2CAP sockets because they possess a stream interface – this can simplify the sending and receiving of data as the client does not need to know the size of the underlying frames RFCOMM uses to exchange data, however, if the protocol being transmitted over the RFCOMM channel is frame-based, it does require that protocol to implement its own framing scheme.

4.2.7 Using Bluetooth sockets

Bluetooth sockets are exposed through the `RSocket` API – however, a specialized Bluetooth wrapper, `CBluetoothSocket` – exists to make using Bluetooth sockets easier. The latter is described more fully later, but this section describes the generic properties of using sockets for Bluetooth.¹¹

¹¹ The use of a socket-like interface for applications allows the state machines of the relevant protocol (L2CAP and RFCOMM for Bluetooth, but also other protocols in Symbian OS) to be hidden away from the application. This contrasts with some other implementations of Bluetooth in the embedded market which provide ‘service interfaces’ to applications: this

One small disadvantage of the `RSocket` API is that it cannot deliver unsolicited protocol indications from the protocol to the application. The most important of these would be a disconnection indication for connection-oriented protocols (such as L2CAP and RFCOMM) – instead, the indication is provided next time the application performs an operation on the socket. `CBluetoothSocket`, being a wrapper over `RSocket`, can provide this service using an internal API – the indications are then delivered through an M-class implemented by the application.

Having said this, applications using `RSocket` directly will find any outstanding operation on a socket such as a write, read or `ioctl` is completed with the system-wide error code of `KErrDisconnected`, or the Bluetooth error of `KErrHCILinkDisconnection` if the channel used by the `RSocket` is disconnected. In practice, since applications will normally maintain an outstanding read on a socket, this means the difference in notification between `CBluetoothSocket` and `RSocket` is merely a matter of convenience.

Socket addresses in Bluetooth are represented by `TBTSockAddr`. It extends the basic `TSockAddr`, described in Chapter 3, with a Bluetooth device address and Bluetooth security settings.

Derived from `TBTSockAddr` are `TL2CAPSockAddr` and `TRfcommSockAddr`. These two classes really only provide typing and casting functionality (although `TL2CAPSockAddr` adds another port setter/getter which can be treated just like those in `TSockAddr`).

Bluetooth Security

Security is obviously a very important part of using Bluetooth – when writing an Bluetooth service, you need to give some thought to your security requirements. The Bluetooth stack is informed of your security requirements in the form of a `TBTSecuritySettings` parameter passed to the `TBTSockAddr`. `TBTSecuritySettings` encapsulates your service's security requirements – such as the need for authentication and encryption – and is used by the Bluetooth stack to apply the requested settings at the appropriate stage of an incoming or outgoing connection.

For those of you interested in the relationship between security as specified in the Generic Access Profile, and that used on Symbian OS, you may be interested to know that Symbian OS operates in security mode 2. For those of you for whom that last sentence made no sense, this means that security requirements such as authorization (i.e., user prompting), authentication (i.e., the process of entering a passkey when

implies that each application which, for example, required an L2CAP connection would need to be aware of the L2CAP state machine in the L2CAP specification. This considerable task is not required for application developers using Symbian OS.

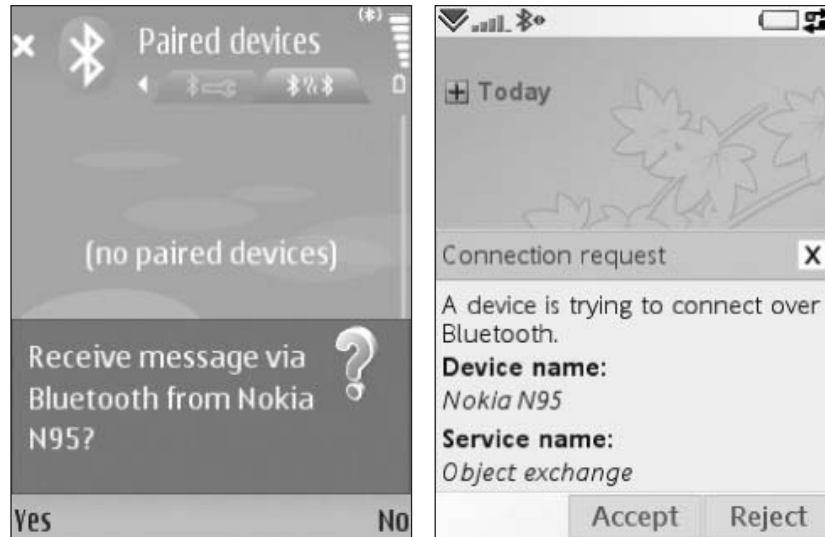


Figure 4.9 S60 and UIQ authorization dialogs

first introducing devices to each other) and encryption are specified on a per-service basis.¹² In practice, this means on a per socket basis.

Careful thought needs to be given to balance usability and security for each service. Both the typical use case and the malicious use case need to be considered. Typically, in order to defend against malicious use of the service, the very minimum a service should do is request authorization for incoming connections. This gives the user the ability to easily reject unexpected connections as a simple 'yes/no' dialog box is presented, as shown in Figure 4.9 (note though that many users will press 'yes' just out of curiosity!). The key issue to consider here is that the device name cannot be trusted as an indication of the source of the connection,¹³ since

¹² The specification also defines security mode 1 (no security) and security mode 3 (security applied when link created). Security mode 1 is obviously too limiting for a device that might run numerous services, each of which will have its own security requirements. Security mode 3 causes problems with usability – it forces users to go through the pairing procedure before allowing a device to perform an SDP query. This can mean that the users perform the pairing procedure only to discover that the service they want to use is not available. Security mode 2 is a reasonable compromise – users know whether a particular service is supported or not before having to perform any security procedures, but individual services can specify their own security requirements. The security risk that is created by security mode 2 is that everyone (including potential attackers) can discover what services are available on your device, however, the increased usability of this option versus the slight trade off in security means that it is a reasonable choice.

¹³ A slightly different dialog is presented if the devices have been paired (and can therefore authenticate each other), indicating that the device connecting is a paired device, which means the name could be trusted. However, the difference is subtle – the addition of the text 'paired device' before the device name. Our opinion is that each service should not rely on the user understanding this subtle difference between the two possible prompts – and

it's easy to set a device name to whatever you like – thus if you need to ensure the connection really is coming from a specific device, use the authentication service described in the next paragraph.

For more sensitive services, such as the ability to send text messages or use some other billable service on the phone, it is recommended that authentication also be required for incoming connections. This will require the user to go through a pairing process, involving entering the same passkey on both devices, to ensure that the two devices are connected to each other and not via some untrusted third party (the classic man-in-the-middle attack). The dialog used for this is shown in Figure 4.10.

Finally, if the data being transferred is in any way sensitive, then it is recommended that encryption is turned on.

For outbound services, it often makes little sense to use authorization, as in most cases the user has just performed some operation that triggers the underlying use of Bluetooth. Thus authorization is a form of 'are you *really* sure?' question. The problem with these dialogs is that they condition the user into pressing 'yes' without considering the question – training the user into this behavior can reduce security significantly, as when the question being asked is actually important they automatically



Figure 4.10 S60 and UIQ passkey dialogs

therefore if it is important to an application to ensure the connection is coming from a particular source, rather than another device potentially masquerading as that source, then it should set the authentication flag on its security settings. Without this the security burden is shifted to the user, who is in a far worse position to understand all the implications of Bluetooth security, pairing and accepting connections from unauthenticated devices than you are!

hit ‘yes’ anyway! It could be argued that outgoing authorization prevents rogue applications creating connections without the user’s knowledge. However, this argument has two flaws – firstly, since it is the application that decides the security settings, then this is easy to bypass – rogue applications simply do not use authorization! Secondly, policing the creation of connections is a role is best left to the Symbian OS platform security model, as this is explicitly designed to restrict this ability to applications with the `LocalServices` capability.

Authentication and encryption on outbound services are more interesting. You need to decide what the user’s expectations are for the security of the data they’re transmitting. In cases where the data is relatively benign, such as when playing a game of Snake, for example, then it is perfectly reasonable not to use authentication or encryption as the data is not sensitive and the additional steps required to use authentication and encryption would reduce usability. However, in the case where you are transferring potentially sensitive information such as the user’s entire phonebook, then it is sensible for the device making the outbound connection to request both authentication and encryption. This means that even if the application on the receiving device has a lax view of security that we have still done our part to protect the user’s data.

There is an additional feature on Symbian OS that is exposed by all UI platforms – the concept of a ‘trusted’ device. Devices can individually be set as trusted in the user interface – as a result of doing this authorization, even if requested by the service, is disabled for these devices – instead the connection is simply accepted. This is a very useful feature for giving an additional level of trust to a set of devices that are owned and controlled by a single user. All of this happens transparently though, so you don’t need to do anything special in your service – the automatic acceptance of incoming connections happens behind the scenes.

Let’s now take a look at `TBTSecurityService` – the interesting methods on this class are:

```
IMPORT_C void SetUid(TUid aUid);
IMPORT_C void SetAuthentication(TBool aPreference);
IMPORT_C void SetAuthorisation(TBool aPreference);
IMPORT_C void SetEncryption(TBool aPreference);
IMPORT_C void SetDenied(TBool aPreference);
IMPORT_C TInt SetPasskeyMinLength(TUint aPasskeyMinLength);
```

The class allows the user to select a combination of authorization, authentication and encryption that is needed for a remote device to connect to the service. The `SetDenied()` method is useful when a service wants to temporarily reject incoming connections, perhaps because it has reached some capacity limit, without having to close its sockets and other resources.

The `SetPasskeyMinLength()` method allows the service to demand passkeys (or PINs as they're often known) of at least a certain length to be used – the resulting token used in the authentication process is stronger when longer pass keys are used.

Principally for the use of the device's Bluetooth control panel, `TBT-DeviceSecurity` allows 'global' security settings for that device to be configured: for example, to set the device as 'trusted' (i.e., needs authenticating, but can bypass any authorization set by a service).

As previously mentioned, all current versions of the Bluetooth specification require the remote device to be authenticated before encryption can be used. This requires the devices go through the pairing procedure, typically on the first connection. The next version of the Bluetooth specification after 2.0 introduces a new, simplified, pairing scheme that is both more usable from the end-user perspective, and allows encryption to be applied without an obvious pairing procedure.

The `SetUid()` method allows for the UI to display an indication of which service is requesting the authorization. This UID should be a standard Symbian UID that is assigned to you, but does not have to be the same as any other UID you are using – however, for ease of administration it might be easiest to use the UID of one of the executables providing the service.

Note that current UI platforms choose not to make use of this information – instead presenting a generic dialog, as per Figure 4.9. However, it is worth passing a sensible value into this method should the authorization prompts change in the future.

Port and security setting: example code

The following code snippet is taken from the example code for this chapter (described in section 4.3). It shows that for this particular service there is only a request to use authorization.

```
TBTServiceSecurity sec;
sec.SetAuthorisation(ETrue);
sec.SetUid(TUid::Uid(KUidBluetoothBusTransport));
// iUpstreamAddress is a TL2CAPSockAddr that cannot be held on the stack
// due to the asynchronous connection call later
iUpstreamAddress.SetSecurity(sec);
// ask the Bluetooth stack to allocate an L2CAP port
iUpstreamAddress.SetPort(KL2CAPPassiveAutoBind);
```

Example 4.8 Setting the security requirements for an incoming connection

More complex security setting

Individual services can themselves nominate individual devices to have different security clearance for connection to that service, although this is typically not recommended as it does not fit well into the way that UI platforms manage settings for Bluetooth devices, which are grouped together to allow the user to control all settings for a Bluetooth device from a single place.

TBTServiceSecurityPerDevice makes this possible – it is simply a container for the Bluetooth address of the remote device for which special security applies, together with the security settings. Each attribute can be set through construction or setters:

```
IMPORT_C TBTServiceSecurityPerDevice(const TBTDevAddr& aDevice, const
    TBTDeviceSecurity& aSecuritySettings);
IMPORT_C void SetDeviceSecurity(const TBTDeviceSecurity&
    aSecuritySettings);
IMPORT_C void SetAddress(const TBTDevAddr& aAddress);
```

To set an override into the listening socket the application puts TBTServiceSecurityPerDevice into a package descriptor (a convenience typedef, TBTServiceSecurityPerDeviceBuf, helps here) and calls SetOpt(KBTSecurityDeviceOverride, KSolBtL2CAP, btSecurityPerDeviceBuf) on the socket to deliver the security settings to the protocol.

CBluetoothSocket

The CBluetoothSocket class brings together into a convenient package the services of RSocket and RBTPhysicalLinkAdapter. CBluetoothSocket supports both the L2CAP and RFCOMM protocols.

The interface to CBluetoothSocket is intended to follow that of the RSocket and RBTPhysicalLinkAdapter. As you'd expect, being a C-class the rules for creation, deletion and copying differ from the R-classes that would otherwise be used. In a similar way to RSocket, not all of the methods presented on CBluetoothSocket are meaningful for the Bluetooth protocols currently implemented in Symbian OS. For example, the connectionless read and write methods (SendTo and RecvFrom) will return KErrNotSupported; however, once Symbian OS supports connectionless L2CAP then these methods will be supported on both CBluetoothSocket and RSocket.

An advantage of CBluetoothSocket is that it removes the need to write the usual set of active object wrappers around RSocket. Whilst CBluetoothSocket is not derived from CActive, it aggregates several active objects to perform operations on an internally held RSocket. Alternately, you might like to consider the wrapper classes shown in

Chapter 3 – although these do not bring the benefit of providing the `RBTPhysicalLinkAdapter` methods in the same class.

Callbacks from the class are made through an M-class interface rather than through a `TRequestStatus&` belonging to the application – therefore the client of the `CBluetoothSocket` is expected to provide a reference to a `MBluetoothSocketNotifier` which it implements internally.

MBluetoothSocketNotifier

```
virtual void HandleConnectCompleteL(TInt aErr) = 0;
virtual void HandleAcceptCompleteL(TInt aErr) = 0;
virtual void HandleShutdownCompleteL(TInt aErr) = 0;
virtual void HandleSendCompleteL(TInt aErr) = 0;
virtual void HandleReceiveCompleteL(TInt aErr) = 0;
virtual void HandleIoctlCompleteL(TInt aErr) = 0;
virtual void HandleActivateBasebandEventNotifierCompleteL(TInt aErr,
    TBTBasebandEventNotification& aEventNotification) = 0;
```

This class is implemented by a user of `CBluetoothSocket` – it is not used if the client accesses Bluetooth protocols via `RSocket`. Note that although the methods have a trailing `L` allowing for the client to leave, the event publisher within Symbian OS does not do anything other than silently consume the leave reason.

As with many objects publishing events through M-classes, the ability to delete the event publisher (in this case `CBluetoothSocket`) within the context of an upcall into the M-class is not supported.

This class allows the client to receive unsolicited indications from the protocol due to internal code in Symbian OS that is not accessible through the underlying `RSocket` class.

4.2.8 Service Discovery

SDP Server

In this section we'll look at how Bluetooth services advertise themselves to remote Bluetooth devices using the SDP server. Before reading this section, make sure you have a good understanding of the concepts in section 4.1.6, as SDP is a complex topic, and the APIs can be confusing if you don't understand what they're doing!

The SDP server in Symbian OS is itself a server in the Symbian OS sense; and is accessed via a number of client APIs.

A session is formed between the application and the SDP server using the `RSdp` class. Typically one service will have one session on the SDP server, although this is not a hard limit but a recommendation to reduce the resources consumed by the service.

```
RSdp sdpSession;
User::LeaveIfError(sdpSession.Connect());
```

Example 4.9 Connecting to the Symbian OS SDP server

Once a session is created, `RSdpDatabase` needs to be opened – this is the mechanism through which records can be added to, deleted from or manipulated in the database.

When registering in the SDDB, an application will typically register a single service record. The service record has two fundamental attributes – the record handle, which is created automatically by the SDP server, and the service class, which is passed into the API at creation time. Here are the methods to register a new service record:

```
IMPORT_C void CreateServiceRecordL(const TUUID& aUUID,
                                   TSdpServRecordHandle& aHandle);
IMPORT_C void CreateServiceRecordL(CSdpAttrValueDES& aUUIDList,
                                   TSdpServRecordHandle& aHandle);
```

Note that you can either use a UUID or a DES (which in this case would contain a hierarchical structure of UUIDs) to identify your service. It is recommended that you use a 128-bit UUID here – you can create one as discussed in section 4.1.6.

The second version of `CreateServiceRecord()` taking an DES offers a more advanced use of the service class field, but using it brings a serious increase in complexity in terms of implementing your service. It allows you to create services that exhibit some degree of ‘service polymorphism’, where the same service can be offered in both simple and (a number of increasingly) complex forms through the same service record. We’re not going to cover that here.

Note that the service must delete the records it has added into the SDDB prior to closing its handles with the server.

Now we can look at the methods on `RSdpDatabase` used to add attributes to the service record we’ve just created:

```
IMPORT_C void UpdateAttributeL(TSdpServRecordHandle aHandle,
                              TSdpAttributeID aAttrID, CSdpAttrValue& aAttrValue);
IMPORT_C void UpdateAttributeL(TSdpServRecordHandle aHandle,
                              TSdpAttributeID aAttrID, TInt aUIntValue);
IMPORT_C void UpdateAttributeL(TSdpServRecordHandle aHandle,
                              TSdpAttributeID aAttrID, const TDesC16& aDesCValue);
IMPORT_C void UpdateAttributeL(TSdpServRecordHandle aHandle,
                              TSdpAttributeID aAttrID, const TDesC8& aDesCValue);
IMPORT_C void DeleteAttributeL(TSdpServRecordHandle aHandle,
                              TSdpAttributeID aAttrID);
IMPORT_C void DeleteRecordL(TSdpServRecordHandle aHandle);
```

In Symbian OS v9.2 the `DeleteRecordL()` method has been deprecated in favour of the newly published `DeleteRecord()` method. The latter is preferable as it removes the potential Leave in awkward places in your application.

Note that the SDP server performs no syntax checking of the structure of the attributes in a record; so when you are registering your SDP record you need to be aware of the syntax of each of the attributes being stored. The SDP specification should be consulted for the syntax of the well-known attributes.

And, as mentioned in section 4.1.6, you can create user-defined SDP attributes (with an ID above 0x0200) and assign any meaning you wish to them – however, it's not uncommon to discover that you don't actually need any of your own attributes at all, in which case there's no need to use them.

When registering an SDP record that advertises a service that can receive connections you should register at least two attributes, although, as described above, the first is done for you as part of creating the service record:

- *ServiceClass* (attribute ID = 0x0001) (or *Service* (attribute ID = 0x0003); though normally only *ServiceClass* is used).
- *ProtocolDescriptorList* (attribute ID = 0x0004).

The *ProtocolDescriptorList* contains information that allows a remote device to configure its Bluetooth protocol stack to connect to your service on the local device. It consists of a list of protocols and protocol 'details'. Each of the common protocols in Bluetooth has been assigned a 16-bit UUID which can be used in the *ProtocolDescriptorList*. These values can be found on www.bluetooth.org/assigned-numbers/service_discovery.php; but the more common are shown below:

- L2CAP: 0x0100
- RFCOMM: 0x0003
- OBEX: 0x0008.

In the case where RFCOMM is used, the well-known L2CAP port on which RFCOMM runs is not required in the *ProtocolDescriptorList*.

For an L2CAP-based service the developer would register the L2CAP port on which its service is running in the *ProtocolDescriptorList* as a 16-bit value.

So, as an example, let's see what the basic SDP registration looks like (in high level abstract terms) for a Symbian OS device acting as a Bluetooth modem (i.e., implementing the DUN profile).

The *ServiceClass* is {*DialupNetworkingUUID*, *GenericNetworkingUUID*}. This is a case where a list of service classes are used rather than just a single one. In this case the *GenericNetworkingUUID* adds nothing of benefit to the *ServiceClass*, so could easily be excluded (indeed, it is marked as optional in the spec).

The *ProtocolDescriptorList* could be, as an example, {L2CAP, RFCOMM, 13}, if the next available server channel at the time the DUN code requested one happened to be 13. In this case, we do not need to give a port for L2CAP, as the RFCOMM port is well known.

It should now be fairly obvious which details you require about your service to register a protocol descriptor list with the appropriate information in – either L2CAP port if you're running over L2CAP, or RFCOMM server channel number, if you're running over RFCOMM. You can see an example of how to do this in section 4.3.5.

SDP agent

Now we've covered registering a service in the local SDDB, it's time to cover searching for a service in a remote SDDB. To avoid confusion between Symbian OS 'clients' (with respect to servers in Symbian OS) and the Bluetooth SDP 'client' of the specification, the term 'agent' is used to refer to the entity that can probe the SDDB of remote devices.

The class that enables use of the SDP agent is *CSdpAgent*; of which the interesting methods are:

```
IMPORT_C void SetRecordFilterL(const CSdpSearchPattern& aUUIDFilter);
IMPORT_C void SetAttributePredictorListL(const CSdpAttrIdMatchList&
    aMatchList);
IMPORT_C void NextRecordRequestL();
IMPORT_C void AttributeRequestL(TSdpServRecordHandle aHandle,
    TSdpAttributeID aAttrID);
IMPORT_C void AttributeRequestL(TSdpServRecordHandle aHandle, const
    CSdpAttrIdMatchList& aMatchList);
IMPORT_C void AttributeRequestL(MSdpElementBuilder* aBuilder,
    TSdpServRecordHandle aHandle, TSdpAttributeID aAttrID);
IMPORT_C void AttributeRequestL(MSdpElementBuilder* aBuilder,
    TSdpServRecordHandle aHandle, const TSdpAttrIdMatchList& aMatchList);
```

To use *CSdpAgent*, a client must implement *MSdpAgentNotifier* to receive callbacks from the SDP agent as it retrieves information from the remote SDDB, and a Bluetooth device address. The interesting methods of *MSdpAgentNotifier* are:

```
virtual void NextRecordRequestComplete(TInt aError, TSdpServRecordHandle
    aHandle, TInt aTotalRecordsCount)=0;
```

```
virtual void AttributeRequestResult (TSdpServRecordHandle aHandle,
                                     TSdpAttributeID aAttrID, CSdpAttrValue* aAttrValue)=0;
virtual void AttributeRequestComplete (TSdpServRecordHandle, TInt
                                     aError)=0;
```

Once an SDP agent is constructed, an SDP query to a remote device involves the following steps:

1. Create a search pattern.
2. Ask for the next record.
3. When records matching the search pattern are found, ask for attributes on those records as necessary.
4. Example code showing the use of the SDP agent APIs is in section 4.3.5.

CSdpSearchPattern

This class encapsulates a list of SDP UUIDs for which to search. When searching with a list of UUIDs, 'AND' search logic is used. However, there is an oddity in the SDP specification – when a client issues a service search or service search attribute request with a set of UUIDs to a remote SDP server, the remote server is allowed to return any service record in which *all* of the UUIDs appear *in whichever attributes*. This is slightly unsatisfactory since the client will usually wish to look for the UUID being listed in the *ServiceClass* attribute. Therefore, a two-phase approach is often made: search for the UUID; receive the list of record handles, then for each record, query the attributes of interest to see if the UUID is indeed in that attribute.

Attribute values in symbian OS

All SDP attribute values in Symbian OS are derived from *CSdpAttrValue*, a common base class which allows attributes of all types to be used polymorphically. You can use *CSdpAttrValue*'s *Type()* function to find the actual type being used and cast appropriately. The inheritance tree is as shown in Figure 4.11.

Useful methods on the base class are:

```
virtual TInt Uint() const;
virtual TInt Int() const;
virtual TBool DoesIntFit() const;
virtual TInt Bool() const;
virtual const TUUID &UUID() const;
virtual const TPtrC8 Des() const;
```

Only those that are meaningful to the derived class are overridden by them; for example *CSdpAttrValueBoolean* will override virtual *TInt Bool()*.

cd Logical Model

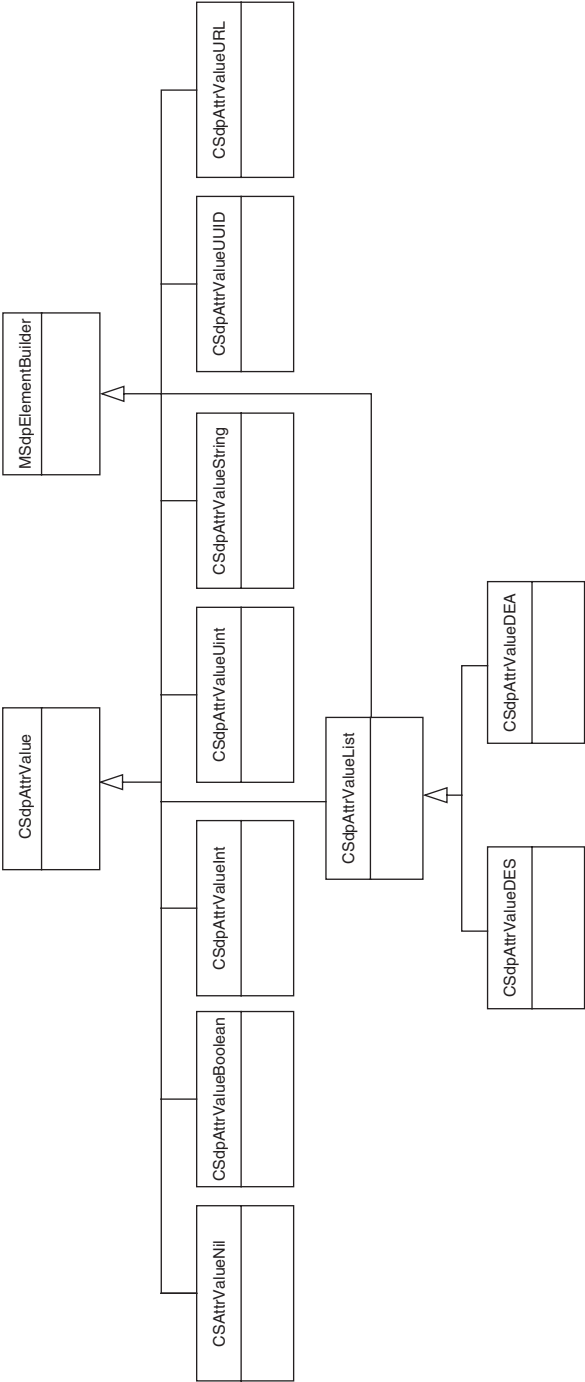


Figure 4.11 Class structure for representing SDP attribute values

`CSdpAttrValueDES` and `CSdpAttrValueDEA` encapsulate the data element sequence (DES) and data element alternate (DEA) types. As previously stated, a DES is a list of values, all of which are relevant; a DEA is a list of values from which one should be chosen. Although the `CSdpAttrValueList`-derived classes are not abstract in C++ terms, they are conceptually abstract, as they are just lists of other `CSdpAttrValue` objects.

It can be seen from Figure 4.11 that `CSdpAttrValueList` implements the `MSdpElementBuilder` mixin. ‘Builder’ here refers to the builder pattern in the GoF book (Gamma, Helm, Johnson, Vlissides 1995), and allows the construction of the complex list-based SDP values: the DES and DEA. SDP allows attributes to comprise of lists, which may themselves contain lists, which may in turn contain another list, and so on – the builder pattern simplifies to some extent the construction of the potentially complex SDP values.

The SDP agent allows the client to specify a set of attributes that it will retrieve from the remote device. The attributes do not have to be contiguous. The `CSdpAttrIdMatchList` class should be used to build the set of attributes. The attributes or ranges of attributes are held in `TAddrRange` structs; and `CSdpAttrIdMatchList` is a container of instances of `TAddrRange`. The SDP agent does not possess an explicit connect or disconnect API – SDP connections are transparently created and destroyed as necessary.

4.2.9 SDAP

The service discovery application profile (SDAP) is not directly supported with its own API in Symbian OS. One particular service present in SDAP is quite useful – the ability to search for a given service on multiple Bluetooth devices. In other words, you can choose to find a service, without caring which device it is running on: most Bluetooth usage scenarios involve the user nominating a device, and then finding the appropriate service on that device to which to connect.

The example developed in this chapter of a multiplayer game uses SDAP.

4.2.10 Registry

Symbian OS provides a permanent store to support the UI and the Bluetooth stack, but also has some use for Bluetooth applications. The store is called the Bluetooth registry. It has a number of internal tables:

- One for persisting the state of the local Bluetooth device, so that settings such as discoverability and the name of the device are preserved between reboots.

- One for details of remote Bluetooth devices, such as a ‘friendly’ name that the user of the local device has assigned to a remote device, and security tokens for remote devices.
- One for settings of the virtual Bluetooth serial ports. All paired devices are stored by the Bluetooth stack in the registry.

The Bluetooth registry is a Symbian OS server – you can access it through a combination of the `RBTRegServ` and `RBTRegistry` APIs.

Once connected, the client can find details of individual devices, or obtain a ‘view’ on the registry’s contents. The view will contain 0 or more devices, constrained by some search criteria. A view is created by using the `RBTRegistry` class which forms a subsession on the registry server. A view is created asynchronously – and the `TRequestStatus` used in the asynchronous request returns with either an error, or, in the successful case, the number of devices in the view.

Note that a value of ‘zero’ represents a successful view creation but which resulted in zero devices in the view.

It is worth briefly surveying the classes that correspond to Bluetooth devices in Symbian OS. The primary class is `CBTDevice`. If Bluetooth names are not important – as is often the case in the Bluetooth stack – a ‘cut down’ version of this class, `TBTNamelessDevice` is also available.

It can be queried with an API that can provide a view onto the registry’s database. To retrieve all of the details of each device in the view, the `CBTRegistryResponse` class is used. It is an active object that will asynchronously create an array of `CBTDevice` classes. For example, a view of all trusted laptops can be obtained, thus:

```
{
// connect a session with Registry Server
User::LeaveIfError(iRegistryServer.Connect());
// Get a 'View' (subsession) with Registry
User::LeaveIfError(iView.Open(iRegistryServer));
// set view to be of Bluetooth laptops held in Registry
TBTDeviceClass laptop(0, EMajorDeviceComputer,
    EMinorDeviceComputerLaptop);
// keep copy of the search pattern to go async
iSearchCriteria.FindTrusted(ETTrue);
iSearchCriteria.FindCoD(laptop.DeviceClass(),
    static_cast<TBTDeviceClassSearch>(EMajorDevice | EMinorDevice));
iView.CreateView(iSearchCriteria, iStatus);
SetActive();
}
// snip - in RunL
...::RunL()
{
    // the iStatus now contains an error, or the number of results in the
    // view
```

```
iResponseHandler = CBTRegistryResponse::NewL(iView);
iResponseHandler->Start(iStatus);
SetActive();
}
```

Example 4.10 Searching the Bluetooth registry for all devices with a minor device type of ‘laptop’

4.2.11 Error Codes

Bluetooth applications should expect errors that are in both the system-wide error range of Symbian OS and a Bluetooth-specific range. The exported header `bt_sock.h` shows the values that are possible. The following are particularly important:

```
Page timeout, KHCIErrorBase-EPageTimedOut, -6004
```

This indicates that the device to which a connection has been requested is either not in range, or is not listening for connections. Some services may wish to treat this as a non-fatal error, and move on to attempt to connect to the ‘next’ device (it is up to the service to deduce what is meant by ‘next’ device).

```
Link disconnection, KErrHCILinkDisconnection, -6305
```

This error is returned if the physical link is unexpectedly lost. This can indicate that the remote device has moved out of range (rather than the user of the remote device instructing it to gracefully close a higher layer protocol connection, in which case the error seen by the local socket user would be `KErrDisconnected`).

```
Access Denied, KErrL2CAPAccessRequestDenied, -6311
```

This error indicates that during a connection attempt the remote device rejected the connection due to the user of that device either rejecting an authorization or the user type a passkey that did not match that entered at the local device.

```
Reflexive link, KErrReflexiveBluetoothLink, -6452
```

The application is attempting to connect to itself: this indicates that the application has got the remote and local Bluetooth device addresses transposed. At present the Symbian OS Bluetooth stack does not support ‘loopback’ connections, so the stack errors the connection with this value.

4.3 Example Symbian OS Bluetooth Application

4.3.1 Overview

We'll now discuss an example application of Bluetooth; that of an ad-hoc non-star-topology transport that can be reused for games, messaging, etc.

The example here demonstrates use of searching for devices and services; forming connections, exchanging data; and managing the Bluetooth topology and using sniff mode.

- `CBluetoothSockets` (and therefore demonstrates how `RSocket` and `RBTPhysicalLinkAdapter` can be used)
- SDP server
- SDP agent
- Device discovery (coupled with SDP to form SDAP).

The intent of the application is to enable 'long-distance' Bluetooth connections; longer than the regular 10 or 100 m Bluetooth link. Use cases for this might include:

- Repeater stations, for example to form a link to a remote printer which is of a distance greater than that reachable by a direct link; but can be reached by making use of intervening Bluetooth stations.
- Gaming – multiplayer 'very wide screen' game – for example a rat runs 'along' all of the devices' and is rendered on each device in turn; the owner of each device has to attempt to trap the rat before it moves onto the next station.
- Messaging – enumeration of all of the Bluetooth devices forming the connection 'substrate' would allow the sending of a message from one station to another.

The example provides the connection topology – the 'bus'. The bus is composed of L2CAP connections over alternating master → slave and slave → master physical links. Over this substrate is the most basic multiplexing protocol – this allows a multiplicity of higher protocols to share the bus. Some of these protocols might provide forwarding functionality, bus enumeration etc.

4.3.2 Implementation

As we have seen in this chapter the basics of a Bluetooth service are to:

1. Register the service record – `RSdp`, `RSdpDatabase`, `CSdpAttr-Value`

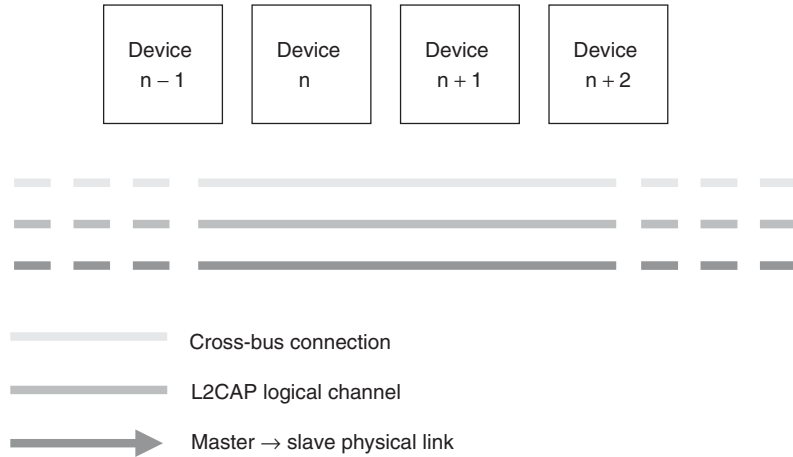


Figure 4.12 Bluetooth link architecture for example application

2. Open listening sockets – CBluetoothSocket, TBTServiceSecurity
3. Find a remote device that can take part in the service – RHostResolver, TInquirySockAddr, CSdpAgent
4. Connect to the remote – CBluetoothSocket, TL2CAPSockAddr
5. Transfer data – CBluetoothSocket,
6. Modify the physical link as appropriate – CBluetoothSocket

Step 3 is exactly the `ServiceSearch` feature of the SDAP profile; which can be made into a useful, reusable combination of `RHostResolver` and `CSdpAgent`. The example code provides a standalone SDAP DLL, and this is used by our example service to find partner devices.

4.3.3 Design

The design yields a reusable ‘Bluetooth bus’ library, an example multiplexing protocol, and an example application protocol (see Figure 4.13).

4.3.4 Service Registration

Recognizing that the service that we have implemented allows incoming connections, the first stage is to register details of the service with the SDP server. This then allows any remote device to see that we have the service available on the local device (remember that we cannot use well-known port numbers in Bluetooth), and discover some dynamic details about the service.

When creating a new service, that is not an implementation of a new Bluetooth profile, a 128-bit UUID is required. For this we just used an

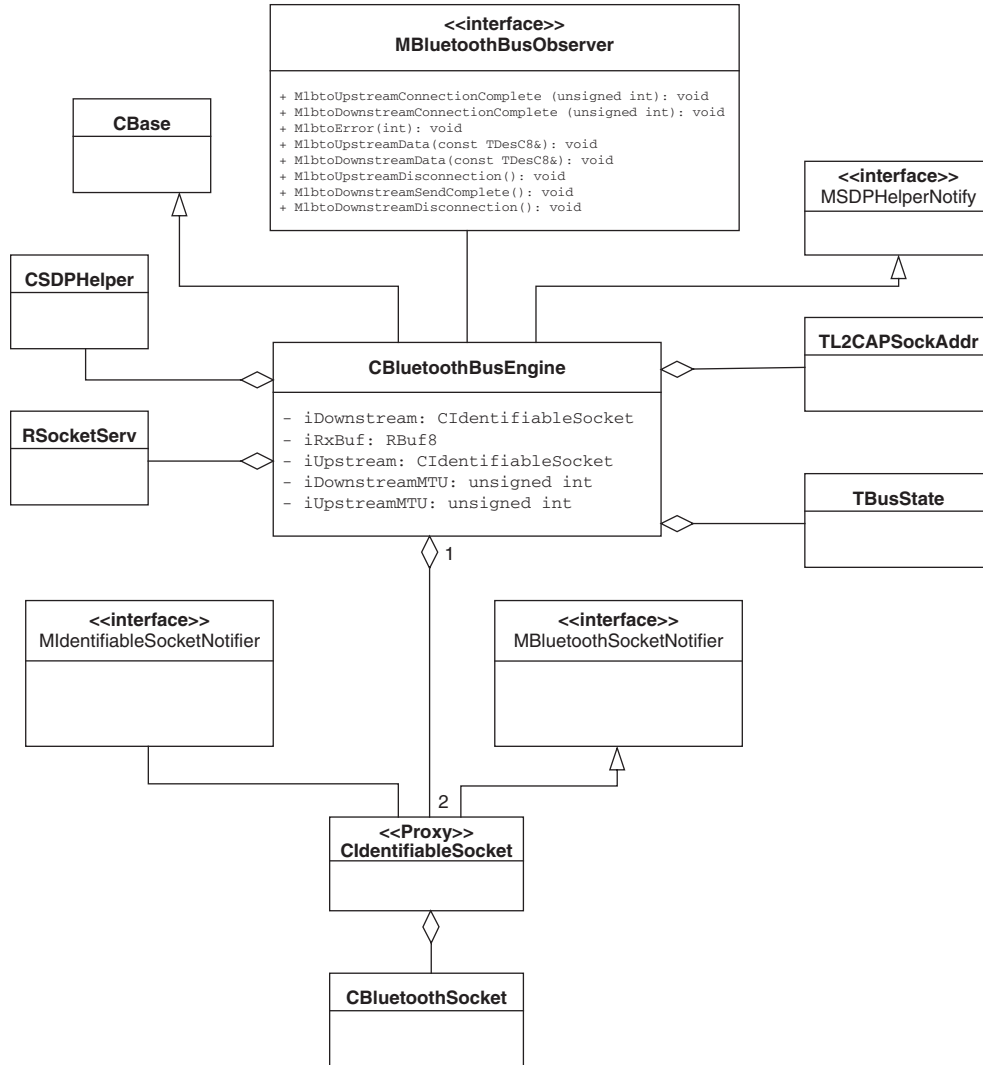


Figure 4.13 UML class diagram for example application

Internet site (for example, see <http://www.itu.int/ITU-T/asn1/uuid.html>) to generate one. After creating a session and subsession with the SDP server, this UUID is then stored in the service record as shown below:

```

// connect to SDP Server
User::LeaveIfError(iSdpSession.Connect());
User::LeaveIfError(iSdpRecords.Open(iSdpSession));
// Set Attribute 1 (service class list) to list with UUID
iSdpRecords.CreateServiceRecordL(TUUUID(KBluetoothBusUUIDHH,
    KBluetoothBusUUIDHL, KBluetoothBusUUIDLH,
    KBluetoothBusUUIDLL), iRecordHandle);

```

The next important attribute to consider is the `ProtocolDescriptorList`. The example application runs directly over L2CAP, so the value is relatively simple:

```
// Protocol Descriptor List
// attrValDES is a CSdpAttrValueDES*
attrValDES = CSdpAttrValueDES::NewDES(0);
CleanupStack::PushL(attrValDES);
attrValDES->StartListL() //this begins the list of protocols
    ->BuildDES(0) //each of which is a DES
    ->StartListL() //which is of course a list
    ->BuildUUIDL(TUUIID(TUint16(KL2CAPUUID))) // L2CAP
    ->BuildUinTL(TSdpIntBuf<TUint16>(aUpPSM))
    ->EndListL() //finished the list for L2CAP
    ->EndListL(); //finished the whole protocol stack
//store this attribute onto the service record
iSdpRecords.UpdateAttributeL(iRecordHandle,
    KSdpAttrIdProtocolDescriptorList, *attrValDES);
CleanupStack::PopAndDestroy(attrValDES);
```

Finally, we use the `ServiceAvailability` attribute to allow a remote to query whether it is worth connecting to the local service:

```
// when registering there is full availability
TUint16 availability = 0;
attrVal = CSdpAttrValueUinT::NewUinTL(TSdpIntBuf<TUint16>(availability));
CleanupStack::PushL(attrVal);
iSdpRecords.UpdateAttributeL(iRecordHandle, KSdpAttrIdServiceAvailability,
    *attrVal);
CleanupStack::PopAndDestroy(attrVal);
```

The above code snippet, although it assigns a value to the `ProtocolDescriptorList` attribute, is enough to demonstrate how other attributes can be set.

4.3.5 Finding the Example Service on Remote Devices

The example application attempts to automatically build large Bluetooth networks. To make the process automatic the device and service discovery procedures are combined into the service discovery application profile; which is provided with the example application.

The interesting part to look at is the code parsing SDP service records and attributes obtained from the remote device.

In the SDAP example, when an SDP record matching the UUID search pattern is found, the following method is called (note the method is defined in `MSdpAgentNotifier`, and implemented in the example by `CSdapEngine`):

```
void CSdapEngine::NextRecordRequestComplete(TInt aError,
    TSdpServRecordHandle aHandle, TInt aTotalRecordsCount)
```


Within this method, if the client of SDAP has asked that SDAP should retrieve attributes from the record (the client would have supplied a non-NULL `CSdpAttrIdMatchList` pointer) an attribute request is performed:

```
// iMatchList is a client supplied instance of CSdpAttrIdMatchList
// containing attributes that should be retrieved from the record found
iSdpAgent->AttributeRequestL(aHandle, *iMatchList);
```

For each attribute retrieved the following method is called (note that the method is defined in `MSdpAgentNotifier`, and implemented in the example by `CSdapEngine`):

```
void CSdapEngine::AttributeRequestResult(TSdpServRecordHandle,
    TSdpAttributeID aAttrID, CSdpAttrValue* aAttrValue)
{
    // tell the client of SDAP
    iNotify.MsanAttributeObtained(iPotentialDevices[iDeviceIndex], aAttrID,
                                *aAttrValue);
    // we still own the SDP attribute, and we're now finished with it
    delete aAttrValue;
}
```

Once all attributes have been retrieved, `MSdpAgentNotifier::AttributeRequestComplete()` is called, SDAP uses this to tell its client that its task is complete.

Now, look at what the example application, as the client of SDAP, does with the `ProtocolDescriptorList` SDP attribute. The application knows the structure of the remote SDP record since the record is defined and published by itself. So, it asserts that the only attribute that is a DES is the `ProtocolDescriptorList`. Then it can proceed to interpret the `ProtocolDescriptorList` to find the dynamic L2CAP PSM (again, the application knows that the service is running on L2CAP, not, for example, RFCOMM. This compromises the ability to change the SDP record. It is recommended that services fully parse the `ProtocolDescriptorList` so that the new versions of the service can easily be deployed, or consult versioning attributes of your SDP record).

Here is the code which begins parsing the `ProtocolDescriptorList`:

```
if (aAttrValue.Type() == ETypeDES)
{
    ASSERT(iAttributeToParse == KSdpAttrIdProtocolDescriptorList);
    // not expecting any DEAs in the record/attribute set we asked for
    //static_cast<CSdpAttrValueDES&>(aAttrValue).AcceptVisitorL( *this);
}
```

The visitor to the attribute operates synchronously, and will call back `MSdpAttributeValueVisitor::VisitAttributeValueL()`

when each value in the list has been visited. In the example the method that implements this callback is as follows:

```
// to check that we have coded to our design correctly
ASSERT(iAttributeToParse==KSdpAttrIdProtocolDescriptorList);
if ((aType == ETypeUUID) && (aValue.UUID() == TUUID(KL2CAPUUID)))
{
    iL2CAPFound = ETrue;
}
if ((aType == ETypeUint) && iL2CAPFound)
{
    // here's the l2cap port!
    // iPSM is a member variable so that we can pass the value out of this
    // visiting loop
    iPSM = aValue.Uint();
}
```

After the ‘visitation’ of the ProtocolDescriptorList is complete, the iPSM value can be put into a TL2CAPSockAddr, along with the Bluetooth device address. This can then be given to the part of the example application that connects L2CAP sockets:

```
if (iL2CAPFound)
{
    TL2CAPSockAddr l2Addr;
    l2Addr.SetBTAddr(aRemoteDevice);
    l2Addr.SetPort(iPSM);
    iNotify->MshnDeviceFound(l2Addr);
}
```

4.3.6 Sockets in the Example Code

The example code makes extensive use of CBluetoothSocket: it shows how they can be connected, how to transfer data, and how to perform physical-link level role switching.

One shortcoming of the CBluetoothSocket class is that if there exists a class that wishes to own two or more of them (as the CBluetoothBusEngine class does), then in the context of a MBluetoothSocketNotifier callback from the CBluetoothSocket, the owning class will not be able to deduce which of the sockets is calling it back.

To solve this problem the example code inserts a new class, CIdentifiableSocket, between the owning class and the CBluetoothSocket, and publishes a new mixin, MIdentifiableBluetoothSocketNotifier, to replace the MBluetoothSocketNotifier. The CIdentifiableSocket implements the regular MBluetoothSocketNotifier; so for each callback from MBluetoothSocketNotifier it forwards the call to MIdentifiableBluetoothSocketNotifier but with a reference to itself. This enables the owner of a

number of `CIdentifiableSockets` to deduce the source of a notification. The `CIdentifiableSocket` also provides a simple getter method to enable the client to access the `CBluetoothSocket` for the purpose of calling the regular `CBluetoothSocket` methods (it would perhaps have been preferable to provide forwarding functions on `CIdentifiableSocket`).

4.3.7 Other Design Points

Within the bus transport layer the state pattern is used to reflect the different states of the uplink and downlink.

The role switches are made in a ‘prospective’ manner; that is if they don’t work then the code will not raise an error – it will continue to provide a service, but possibly the service is not optimized. The role switch is sought so that the device tries to ‘scatternet’ – that is be a master and a slave at the same time. When the ‘downstream’ socket is connected, the physical link over which the socket runs is requested to be master at the local device. Conversely, when the ‘upstream’ socket is connected, the physical link over which that socket runs is requested to be slave at the local device:

```
void CInfiniBTEngine::DoDownstreamConnectionComplete(TBool
    aUpstreamPresent)
    ....
    (void)iDownstream->Socket().RequestMasterRole(); //ignore error
    ...
    void CInfiniBTEngine::DoUpstreamConnectionComplete(TBool
        aDownstreamPresent)
    ...
    (void)iUpstream->Socket().RequestSlaveRole(); //ignore error
```

4.4 AV Protocols and Profiles

This section includes a brief discussion about the AV protocols in Bluetooth; even though direct interfaces to the protocols are not provided in Symbian OS. However, the profiles they support can be used via high-level abstraction APIs.

The audio video control transport protocol (AVCTP) is a message-based protocol that encapsulates media-control frames into L2CAP packets. It provides a fragmentation service to the upper layer so that the higher layer packets can be split over the potentially smaller L2CAP packets. The fragmented frames are reassembled at the receiving AVCTP entity before passing to the upper layer. In effect it is hiding the L2CAP packet size from the layer above AVCTP; thus enabling a legacy implementation of a media-control protocol to not need to be changed to fragment its packets.

The audio video remote control profile (AVRCP) is the only Bluetooth profile that specifies use of AVCTP at present. AVCTP also provides a

limited multiplexing service, so in future other profiles relating to media-control can coexist over the same L2CAP channel. AVRCP captures in a Bluetooth specification the use cases of a user controlling a media device such as a TV, CD player or amplifier: those use cases that are currently dominated by proprietary infrared remote control devices. In the future it is to be hoped that more consumer electronic devices will use Bluetooth for their control needs: this would open up the compelling possibility of using a Symbian OS device to present a remote control user interface of the user's choice, and allow, for example, auto-muting when a phone call arrives.

AVRCP takes advantage of the services of AVCTP, to run an existing media-control protocol via Bluetooth. The media-control protocol specified is a subset of the AV/C protocol that was originally designed to be transported over Firewire (a wired technology that is often used to transport AV data, and control the devices supplying or consuming the data). AVRCP defines a device that is sending an AVRCP command as a 'controller', and the receiver of such a command as a 'target'. AV/C models an AV device as being constituted of a 'unit' that comprises a number of 'subunits'. Each subunit might be for example, a tuner, a mixer or an amplifier. There is also one more subunit in an AV/C unit that is key to the AVRCP subset of AV/C – that is the 'panel' subunit. In version 1.0 of the AVRCP specification, all commands are sent to the panel subunit. However, the commands that are sent are actually known as 'passthrough' commands: it is therefore up to the panel subunit in the 'target' device to route the command to the correct logical subunit. This simplifies the task of the AVRCP controller as it does not have to find out the addressing details of all of the subunits in the target.

For the distribution of AV data itself (rather than the explicit user-level control of the media device), the audio video distribution transport protocol is used. AVDTP, which runs over L2CAP, is both a signalling protocol (in the control plane) and a multiplexing datagram protocol (in the user plane). It is the key protocol used when streaming AV media – it discovers what codecs are available on a remote device, whether they are available and what the codec capabilities are. AVDTP can then be used to negotiate bitrates and other settings that optimize the streaming performance. Finally AVDTP is used for starting and suspending the running of the remote codec; and also to close codec connections.

The generic audio video distribution profile (GAVDP) is a specification that adds details about how AVDTP is used to support AV streaming use cases. It is an 'abstract' profile that does not yield concrete use cases; although it does yield an API in Symbian OS that licensees can use to implement audio or video streaming (the advanced audio distribution and video distribution profiles, respectively). The licensee may elect to add another interface onto the GAVDP interface in Symbian OS so that developers can develop GAVDP-based profiles themselves; however,

shipped as standard, only Symbian partners can develop against the GAVDP API.

The A2DP profile specifies a mandatory codec to support in all Bluetooth devices claiming to support A2DP; it also specifies that the codec data be transported via RTP and AVDTP. This is therefore not that dissimilar to streaming over the Internet which typically has codec data encapsulated in RTP packets, but which are transported via a layer-4 transport datagram protocol (UDP). Of course, the benefit of A2DP is that it recognizes that in most use cases of Bluetooth streaming there is no need for the layer 3 or 4 protocols since the connection from source to sink is point-to-point. This demonstrates one of the clear benefits of Bluetooth profiles: reuse existing technologies when appropriate, but introduce efficiencies as needed, and also introduce mandatory components so that devices have increased interoperability. Note that the mandatory codec ('SBC') is not one of the more common audio codecs: this is to remove any need for codec licensing fees, and also to optimize the use of processing devices in each Bluetooth device. The mandatory codec yields a higher bitrate for the same quality as other codecs, but requires less machine instructions to operate – which leads to a reduced battery consumption.

The video distribution profile (VDP) is very similar to A2DP, but with different codecs specified.

AVDTP also has a data-plane service that allows multiplexing of AV data into more complex sets of L2CAP channels. Although this is implemented in Symbian OS it is not currently used since no other devices currently support the feature.

Notice that although AVDTP has a control-plane aspect, it differs from the service provided by AVCTP and AVRCP: the latter implements the user-level control, whereas the control plane of AVDTP is invisible to the user, and is not directly used when a remote-control button is pressed. Following from this, although it is often the case that a Bluetooth AV device may have AVDTP and AVCTP implemented, this is not a requirement. For example, basic stereo Bluetooth headphones may not implement AVRCP/AVCTP at all; yet another device may be purely a remote control, and thus not implement AVDTP.

The two AV protocols of Bluetooth, AVDTP and AVCTP, are available in Symbian OS from v9.1, but are not directly available for direct, general application use. This is done for good reason: the use cases enabled by these protocols imply that direct access to them is actually *not* useful for applications. However, a framework for sending and receiving remote control signals is provided that can be used by applications.

For the distribution side of Bluetooth AV, no direct access to these APIs is presented. Instead, the GAVDP API is used via the multimedia framework of Symbian OS. In other words, Bluetooth stereo headsets or speakers are rendered in Symbian OS as though they are local audio devices: this has the very compelling feature that any extant multimedia

application can stream over Bluetooth without any change (or indeed knowledge that it is streaming via Bluetooth). Symbian OS licensees own the policy of selecting the appropriate audio input/output devices to send or receive audio.

4.4.1 Remote Control

This remote-control framework, first delivered with the Bluetooth AVRCP profile in Symbian OS and sometimes known as ‘Remcon’, abstracts away the transport (e.g. Bluetooth, IrDA) and the protocol (e.g. AVRCP or proprietary). Therefore applications can expect to receive events of the form ‘Play’, ‘Stop’ rather than requiring to interpret byte representations of remote-control protocols. Each of the transport/protocols combinations is called a ‘bearer’ and the standard Symbian OS includes the Bluetooth/AVRCP bearer. The device manufacturer may include other bearers to support, for example, a wired headset.

The events and commands that are presented via the ‘core’ RemCon API are based on the AV/C specification (upon which AVRCP is based). This specification need not be consulted to use RemCon; but it is noted here as AV/C is a common, generic set of remote control commands.

One interesting issue regarding implementation of AVRCP (and other remote-control bearers) on an open platform, such as Symbian OS, is that of where to send incoming remote-control signals. For example, a ‘Play’ signal arrives from a remote Bluetooth headset: to which application should it be sent? What about volume controls? Should they be sent elsewhere in the system?

The remote-control framework allows Symbian OS licensees to implement a ‘routing’ policy that allows for their particular combination of UI metaphor (for example, whether non-visible applications are still running or not), device-wide volume policy, inbuilt media applications, and potential downloadable applications to be sent remote-control commands. The routing policy also applies to the sending of outgoing remote control in a ‘connectionless’ manner (i.e., the application doesn’t care to which device the command is sent). The use case for these connectionless commands is, for example, a volume-setting user interface informs Remcon of a need to send a volume up signal to the current rendering device. The policy can query the audio policy object to see which device is currently rendering, and forward the volume remote-control signal to that device. The licensees routing policy is encoded into a plug-in inside Remcon.

This section does not provide an example application; however, snippets of code will be presented that demonstrate how a remote-control command can be sent, and received. Following the description of the routing policy, the ability for an arbitrary application to receive events is dependent on the routing policy.

An application that wishes to receive and/or send remote control signals first creates an ‘interface selector’ through instantiation of a `CRemConInterfaceSelector` class. It then attaches classes representing different APIs to the interface selector: the two main APIs are `CRemConCoreApiController` (for controllers) and `CRemConCoreApiTarget` (for targets). Both of the core API classes may be attached to the interface selector simultaneously.

A concrete remote control API is attached to an interface selector through a call to the API’s `NewL()`. The `NewL()` takes as parameters a reference to the interface selector, and a reference to an callback interface mixin published with the remote control API. Therefore the client of (for example) the `CRemConCoreApiController` class must provide a reference to an implementation of the `MRemConCoreApiControllerObserverInterface`.

The `MRemConCoreApiControllerObserver` interface simply provides notification of the result of the sending by the controller of a command (a non-error result indicates the successful sending of the command in the opinion of the appropriate bearer).

The `MRemConCoreApiTargetObserver` is particularly important to applications that expect to receive remote control commands: it is the interface through which commands are delivered. Some commands, such as `Play`, have dedicated APIs through which to deliver command-specific options (in the case of ‘Play’ a parameter, derived from `AV/C`, is supplied which gives the speed at which to play the media). Commands that do not have parameters are all delivered through the `MrcctoCommand` method. (Note that certain M-classes in Symbian OS take a random looking set of letters to prevent name clashing of methods from other M-classes.)

All of the methods in the `MRemConCoreApiTargetObserver` and `CRemConCoreApiController` which relate to the receiving or sending of commands take a `TRemConCoreApiButtonAction` parameter. The `TRemConCoreApiButtonAction` enumeration consists of the values:

- `ERemConCoreApiButtonPress`
- `EremConCoreApiButtonRelease`

ERemConCoreApiButtonClick

For the controller case it allows the sending of either a ‘click’ of a virtual button, or a ‘press and hold’ of the virtual button. The ‘press and hold’ feature allows the user to click a UI widget to begin (for example) increasing the volume of a target device. In this case the application would send the `Play` command with `ERemConCoreApiButtonPress` at the time the UI widget is depressed. When the UI widget is released, the `Play` command would again be sent but with the `TRemConCoreApiButtonAction` value of `ERemConCoreApiButtonRelease`.

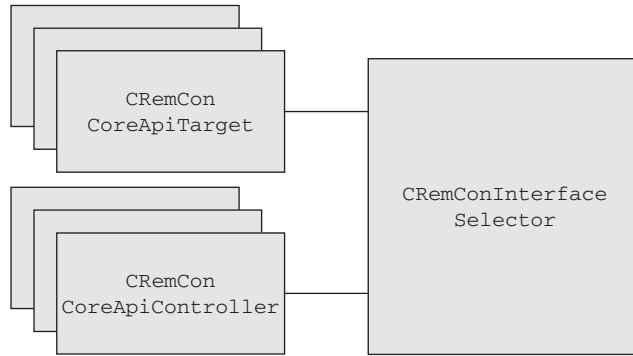


Figure 4.14 Block diagram of client-side remote control classes

In the target case, the bearer attempts to deduce a value for the button action – this is then delivered to the application. The AVRCP specification contains a number of timer values that allow an AVRCP entity to attempt to deduce the button action – even though the packets on the air (or wire in the case of Firewire!) only specify button press or release. The process is not particularly simple for the AVRCP entity – this is one more benefit of hiding AVRCP itself from the application developer.

Target example

The following code shows the construction of the interface selector, and the attachment of a remote control API that allows for the receipt of remote control signals from a remote device.

```

iRemConInterfaceSelector = CRemConInterfaceSelector::NewL();
iRemConTarget = CRemConCoreApiTarget::NewL(*iRemConInterfaceSelector,
    *this);
iRemConInterfaceSelector->OpenTargetL();
//
  
```

Controller example

```

iSelector = CRemConInterfaceSelector::NewL();
iCoreController = CRemConCoreApiController::NewL(*iSelector, *this);
iSelector->OpenControllerL();

//Ask user which device address we should connect to...
TBTDevAddr btAddr;
GetDeviceAddressL(btAddr);
// Store as 8 bit machine readable
RBuf8 bearerData;
bearerData.CreateL(btAddr.Des());
// Form the RemCon Addr from the AVRCP Uid and the
// bluetooth address
TRemConAddress addr;
addr.BearerUid() = TUid::Uid(KRemConBearerAvrcpImplementationUid);
  
```



```
addr.Addr() = bearerData;
iSelector->GoConnectionOrientedL(addr);
iSelector->ConnectBearer(iStatus);
SetActive();

//send Play
case ERemConCoreApiPlay:
    iCoreController->Play(iStatus, iNumRemotes,
        ERemConCoreApiButtonClick);
//send Stop
iCoreController->Stop(iStatus, iNumRemotes, ERemConCoreApiButtonClick);
//send volume control when user presses UI widget
iCoreController->VolumeDown(iStatus, iNumRemotes,
    ERemConCoreApiButtonPress);
// user releases UI widget on touchscreen
iCoreController->VolumeDown(iStatus, iNumRemotes,
    ERemConCoreApiButtonPress);
```

4.5 Summary

In this chapter we have learnt about:

- discovering and connecting to other Bluetooth devices
- searching for services offered by remote devices
- registering our service on the local device
- security on Bluetooth links
- low-power modes on Bluetooth links
- L2CAP, RFCOMM, and the advantages and disadvantages of each
- AV and remote control services provided by Bluetooth protocols and profiles, and the remote control framework.

Reference

Gamma, E., Helm, R., Johnson, R. and Vlissides, J. (1995) *Design Patterns: Elements of reusable object-oriented software*, Addison-Wesley Professional.

5

Infrared

5.1 Introduction

Wireless communication started on Symbian OS with the support for the infra-red protocol stack standardized by Infrared Data Association (IrDA). IrDA provides a networking stack optimized for ad-hoc information exchange, and can be used for many purposes. With the introduction of alternative technologies, however, its major current use case is exploiting the large installed base of IrDA transceivers on laptops and other computers, and for one-off exchanges of small amounts of data. Its primary advantages are speed and simplicity of connection establishment, together with the perception of enhanced security because of the line-of-sight requirement.

In this chapter we examine the IrDA technology and show how the concepts translate to the Symbian OS implementation. We cover device and service discovery, listening and connecting sockets at different levels in the protocol stack, and provide an overview of the socket options and ioctls provided for advanced purposes.

5.2 Infrared Overview

The IrDA standards cover data communication over infrared light at all levels – from hardware to application. This includes details such as range and light cone angle at the hardware level, working up to device discovery and addressing, service discovery, and creating a reliable, multiplexed data link.

Key benefits of the IrDA protocols and associated hardware include very low cost and high levels of maturity, which together have greatly increased market penetration. Until very recently, IrDA was the most popular short-range wireless link on a variety of devices, including

mobile phones, PDAs and laptop computers. Bluetooth was created more recently to fulfil a similar role to IrDA, but infrared still maintains a significant presence in mobile devices – particularly in the Asian market. Key differences between IrDA and Bluetooth include range, line of sight and device discovery and connection setup times (see Table 5.1).

Due to these differences, IrDA protocols still have a useful role to play, particularly when small objects need to be transferred between two devices that will not be frequently connected. However, the most important factor in deciding whether to use infrared is the availability of hardware in products. Whilst many phones in the market have Bluetooth functionality built in, some phones are now omitting the infrared port, especially in European markets.

It is also worth noting that the Bluetooth standards body are considering methods to address the differences in device discovery and connection setup times in relation to IrDA, so devices supporting future versions of the Bluetooth specification are likely to approach the same performance as IrDA.

Table 5.1 Key differences between IrDA and Bluetooth

	IrDA	Bluetooth
Range	1 m	10 cm, 10 m or 100 m (10 m most common)
Line of sight	Required	Not required, although obstacles may reduce range
Device discovery time	<0.5 s	10s of seconds, depending on number of devices found. Pairing removes this delay when repeated connections to the same device are likely
Connection setup time	<0.5 s	<10 s
Speed	Serial Infrared (SIR): up to 115 kbit/s Faster IrDA physical layers are specified but are not supported by Symbian OS	768 kbit/s Enhanced Data Rate (EDR): 1.4 or 2.1 Mbit/s

5.2.1 The IrDA Stack

Figure 5.1 shows a block diagram for an IrDA stack. At the bottom of the IrDA stack is IrPHY – the physical layer. This layer is too low to be of much interest when developing applications using IrDA, but is responsible for defining the angle of the light cone, the range of the beam and the raw link speed.

Moving up a layer is IrLAP – the Link Access Protocol. This layer provides for establishing reliable connections, so includes device discovery and addressing, connection establishment and teardown, reliable data exchange and link level flow control. It is worth emphasizing that this level in the IrDA stack provides the reliability, so all higher layers are automatically reliable.

Next comes IrLMP – the Link Management Protocol. This is divided into two sections: LM-MUX (often referred to as IrMUX on Symbian OS) and LM-IAS. IrMUX provides multiplexing, allowing a number of applications to share the connection. It provides 127 stream endpoints (LSAP_SELs – Logical Stream Access Point Selector, often referred to as *ports*) per device, although 16 of these are multicast, and LSAP_SEL zero is reserved for IAS (Information Access Service). Some IrDA implementations may allow the source and remote LSAP_SEL together to define a channel,

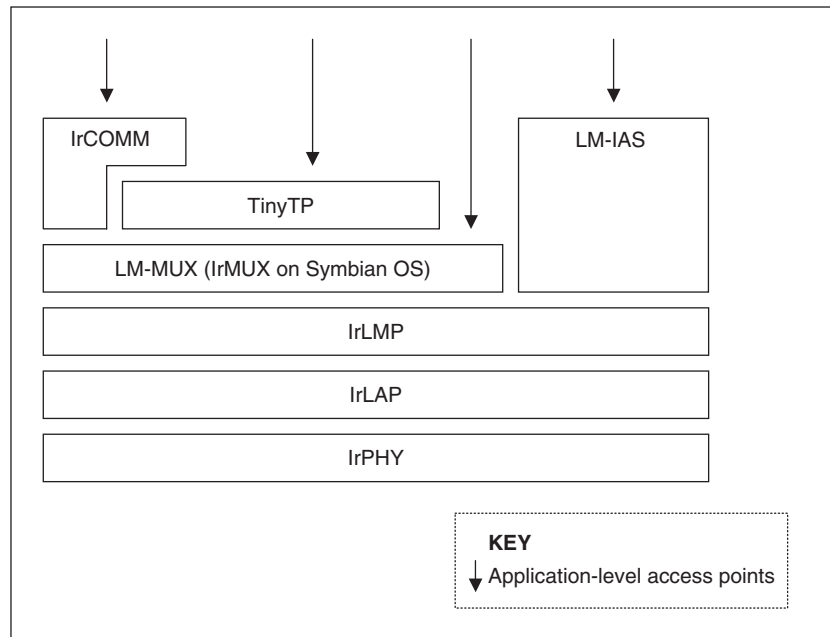


Figure 5.1 IrDA stack

so increasing the number of channels available. IrMUX is the lowest protocol layer exposed on Symbian OS, and is available as a sequenced datagram socket. It is worth noting that, particularly with the flow and error control in IrLAP, it is possible to deadlock an IrDA link if two IrMUX connections are active. It is possible to mark a connection as *exclusive* to prevent this problem – all other links are flow-controlled off. In platform security enabled versions of Symbian OS, i.e. v9.1 onwards, the ability to perform this operation is protected more strongly than standard data transfer.

As availability of IrMUX LSAP_SELs is limited, IrDA profiles tend to use dynamic selection of these identifiers. Unlike HTTP/TCP, which always listens on port 80, OBEX (the IrDA equivalent of HTTP) may listen on any LSAP_SEL, possibly changing each time a device is discovered.¹ To allow connection to these dynamic ports, a device can connect to the single well-known LSAP_SEL zero, and query the IAS database.

IAS queries generally consist of a class name and the attribute in which we are interested for that class. Class names are a simple text string – certain services use certain well-known strings, and other services can choose their own strings. When choosing your own string it is sensible to prefix it with a unique value – your company name, for example – to reduce the chance of clashes. The server then responds with the corresponding value for that attribute. This is easier to see using an example.

	Class name	Attribute
Query	OBEX	IrDA:TinyTP:LsapSel

Figure 5.2 A typical IAS query for the default OBEX service

Figure 5.2 shows a query for information about the default OBEX service, specifically the LSAP_SEL on which it is listening.

Figure 5.3 shows the result, telling us that the service is running on LSAP_SEL 2.

To address the potential deadlocks of IrMUX, Tiny Transport Protocol (TinyTP) is the next layer. TinyTP provides credit-based flow control on a per-channel basis rather than globally across the entire link. It is otherwise identical to IrMUX, other than (in the IrDA specification) allowing for either a stream interface or a sequenced datagram interface. Most IrDA applications should use this layer unless they have a good reason not to.

	Type	Value
Response	Integer	2

Figure 5.3 A typical IAS response for the default OBEX service

¹ Here we refer to the general OBEX service. OBEX services offered by specific applications may also exhibit this behaviour, as they should not be hard-coding a ‘standard’ LSAP_SEL either, for the reasons described in this section.

The final protocol is IrCOMM – serial port emulation. To describe IrCOMM purely as a serial port emulation is oversimplistic, although it is usually exposed to applications through a serial port-like interface. There are multiple ‘levels’ of IrCOMM – IrLPT; 3-wire raw; 3-wire cooked; 9-wire; and Centronics. The names change as functionality increases. IrLPT and 3-wire raw are essentially the same, although they use IAS slightly differently. The biggest drawback is that they run directly over IrMUX, creating the possibility for deadlocks. Neither variant provides emulated control lines, so the RS-232 hardware flow control signals are unavailable. 3-wire raw runs over TinyTP and provides a control channel. Being a three-wire serial port, this control channel still does not carry flow control lines, but it can be used to signal a baud rate (although this is purely advisory – the emulated link will actually run at the physical link speed), communications errors, XON/XOFF characters and the break signal, among other things. For a full list of the control channel data, consult the IrCOMM specification.

The 9-wire variant builds on the 3-wire control set by adding support for changing and querying the RS-232 flow control lines – specifically DTR, DTS, CTS, DSR, RI and CD. Obviously which of these signals are valid depends on whether you are in DTE or DCE mode. It should be noted that as these lines are sent as TinyTP data interleaved with the IrCOMM level data, precise timing should not be relied upon. Some implementations may also choose to ignore the RS-232 level lines, and rely instead on TinyTP’s credit-based flow control. As such it is not possible to assume that having asserted a ‘flow off’ signal line, the remote station will stop sending data.

Centronics support adds still more control channel data to carry the extended set of signal lines, mainly various IEEE1284-defined signal lines.

5.3 IrDA in Symbian OS

5.3.1 IrCOMM

Although IrCOMM is the highest level protocol in the IrDA protocol stack, it is worth mentioning first as the Symbian implementation takes care of all of the IrDA-specific behaviour.

To use IrCOMM, open an `RComm` instance with port name `IrCOMM: : 0`. Remote stations may now connect into the local device, using any of the IrCOMM variants. Writing to the port will attempt to discover and connect to the first remote station that is found. User applications have no control over the variant of IrCOMM used – the highest variant supported by the remote machine is used (starting with 9-wire cooked). IrCOMM also supports encapsulating serial data directly in TinyTP packets – a protocol often referred to as IrNET or IrDIAL. This TinyTP encapsulation will be used in preference to any of the pure IrCOMM variants

where possible. Just as there is no way to select which variant to use, there is equally no way to interrogate an established link and discover the type.

As a result of this automatic negotiation, the RS-232 flow control lines exported by IrCOMM must not be relied upon to be sent to the remote station, or to reflect what the remote station is attempting to assert. For the 3-wire protocols, signals are looped back, so, for example, asserting RTS locally causes the local CTS signal to be asserted.

Another point to note is the use of port configurations. None of the baud rate or similar parameters will affect the real IrDA link, but may be passed to the remote station on a cooked link. Of greater importance is the `iSIREnable` flag. This flag is used to enable IrDA framing on the serial line, which initially sounds relevant to IrCOMM. In fact, as this is an electrical level issue, attempting to set this on an IrCOMM `RComm` instance will lead to `SetConfig()` returning an error, and none of the other parameters supplied being changed.

For applications using IrCOMM, the fact that they are running over the IrDA protocols should be fairly transparent. As a result, the remainder of this chapter is unlikely to be relevant – although monitoring of link status, described in section 5.3.13, may be of interest.

5.3.2 Protocol Constants and API Definitions

As described in chapter 2, the IrDA protocols are implemented on Symbian OS as an ESOCK plug-in. This allows the use of standard `RSocket` interfaces, coupled with the associated framework classes. `RSocket` is defined in `es_sock.h`, and the IrDA-specific classes and constants are defined in `ir_sock.h`.

The most important constants defined in `ir_sock.h` are those relating to the address family and protocol. These constants are required to open any class from `es_sock.h`, and associate it with the IrDA implementation.

```
const TUint KIrdaAddrFamily    = 0x100; // IrDA protocol family
const TUint KIrmux             = 88;    // Protocol number for Irmux
const TUint KIrTinyTP          = 89;    // Protocol number for IrTinyTP
```

5.3.3 Device Discovery

```
class CIrDeviceDiscoverer : public CActive
{
public:
    ...
    void DiscoverDevicesL();
    virtual void RunL();
    // DoCancel would also need implementing for real applications
    ...
}
```

```

private:
    RSocketServ iSs;
    RHostResolver iDevices;
    TNameEntry iResult;
};

_LIT(KAllDevices, "");
void CIrDeviceDiscoverer::DiscoverDevicesL()
{
    // Open a connection to socket server
    User::LeaveIfError(iSs.Open()); // this could be done in ConstructL

    // Open an IrDA host resolver
    User::LeaveIfError(iDevices.Open(ss, KIrdaAddrFamily, KIrTinyTP));

    // Run a discovery sequence
    TInt err = iDevices.GetByName(KAllDevices, iResult, iStatus);
    SetActive();
}

void CIrDeviceDiscoverer::RunL()
{
    if (iStatus.Int() == KErrNone)
    {
        // Do something with the discovered device
        TIrdaSockAddr addr(result().iAddr);
        // on v9.2 only, you could start searching for the next device here
    }
    else
    {
        // Discovery failed.
        User::Leave(err);
    }
}

```

Looking through this snippet illustrates a number of important points. `RHostResolver::Open()` takes three parameters – the socket server handle, the address family and the protocol number. In the case of IrDA only the address family is important, although ESOCK will enforce that the protocol number is one that the IrDA stack claims to know about. In other words, it does not matter whether IrMUX or TinyTP is specified in this call. However, for the sake of clarity it is suggested that the protocol which will eventually be used be specified.

The call to `GetByName()` is shown here using the asynchronous version of the call, as it can take up to around two seconds to complete. If a synchronous call is required in a real application, it would be better to use the two-argument version of `GetByName()` – but as with most Symbian OS programming, it is preferable to use an active object as this means infrared functionality is usable from threads that are also performing UI tasks without causing the UI to freeze.

The other point to note in the `GetByName()` call is the first parameter. This descriptor is passed to the host resolver to tell it which name the programmer is interested in. In the case of IrDA, however, the highly

dynamic behaviour is such that the name makes little sense, and so is ignored by the stack. For the sake of clarity, it is recommended that the asterisk wildcard be used, as is suggested in this code snippet. Using this wildcard will also protect your code should this behaviour change in future releases.

This code snippet only illustrates retrieving the first discovered device. `RHostResolver` provides the function `Next()` to retrieve additional matches, but it should be used with caution on the IrDA protocols. On versions of the stack shipped in Symbian OS v9.1, `Next()` always returns the first discovered device and no error. This has changed in v9.2, which will return additional devices, and complete with an error code when all devices have been found. If you wish to allow your code to handle discovering multiple devices, call `Next()` and verify that the device address returned is new. Due to the lack of an error return value in v9.1, the only way to reliably tell when the discovery sequence is complete is to check whether the address has already been seen.

5.3.4 Making a Device Visible

As long as the stack is loaded, it will respond to discovery requests received from a remote station. Although there are a number of possible ways to load the stack, the most appropriate will be to open a listening socket – without a service listening and ready to accept connections, there is little point in allowing other stations to discover the local device. It is, however, also possible to open a host resolver, a service resolver (`RNetDatabase`), or simply use `RSocketServ` to force loading of the stack. Often services provided over infrared are time-limited – that is to say that a specific action needs to be taken in the UI to activate a listening service, then the service is only active for a limited period – one to two minutes, for example. This is controlled at the service level – the stack will remain running for as long as the service is listening. It is therefore recommended that you include code in your service that limits the period for which it listens for incoming connections.

When visible, the stack responds to discovery requests with the device nickname. The nickname is a short user-friendly name, which may be presented to users if multiple devices are discovered. The nickname has a maximum length of 20 bytes, and is always sent in ASCII encoding. The IrDA specification does allow different character sets to be used, but for interoperability reasons, Symbian OS only allows ASCII.

To set the device nickname, use the `SetLocalName()` function on `RHostResolver` attached to the IR stack. This operation should only be performed by the device manufacturer, and as such, is protected by the `NetworkControl` capability.

There may also be times when suppressing of discovery responses is of use. This may be the case when connecting to an OS such as Windows, where as soon as a device is discovered it is connected to, in order to

interrogate its supported features. Currently the only way to do this is with a socket option (`KDiscoveryResponseDisableOpt`), which is described in more detail in section 5.3.11. Using this option requires a socket – which corresponds to the stack being loaded. As a result, there is a short race window between loading the stack and instructing it to ignore discovery sequences. Note that use of this option requires the `NetworkControl` capability, which makes it unsuitable for general-purpose use. This window is unlikely to be encountered in real situations, but future versions of Symbian OS will offer a Publish and Subscribe key which may be used as well.

5.3.5 Service Discovery

Once a device has been found, IAS can be used to discover what services are running and how to connect to them. On Symbian OS, this functionality is provided by the `RNetDatabase` class. As with all of the other ESOCK interface classes, this generic API is extended with the required IrDA specific concepts by the IrDA stack, and so parameters are passed as a generic descriptor package.

Compared to device discovery, which only has a single specialized class to deal with, service discovery has three. Two are used for discovering services on other devices – `TIASQuery` and `TIASReponse`, and one for registering services with the local IAS database – `TIASDatabaseEntry`. We'll explore discovering services on remote devices first, then look at how to register entries in the local IAS database.

When submitting queries, two classes are used:

```
NONSHARABLE_CLASS(TIASQuery) : public TBuf8<KMaxQueryStringLength>
{
public:
    IMPORT_C TIASQuery(const TDesC8& aClass, const TDesC8& aAttribute,
                      TUint aRemoteDevAddr);

    IMPORT_C TIASQuery();

    IMPORT_C void Set(const TDesC8& aClass, const TDesC8& aAttribute,
                     TUint aRemoteDevAddr);
    IMPORT_C void Get(TDes8& aClass, TDes8& aAttribute,
                     TUint& aRemoteDevAddr);
};
```

`TIASQuery` is used when formulating the query to send to a device. When using this class, the class name and attribute value should be specified either at construction time, or by calling `Set()` just prior to executing the query. A remote device address is also taken – note that this is a simple integer rather than a full `TIrdaSockAddr` class. The device address can be extracted from a `TIrdaSockAddr` by calling `GetRemoteDevAddr()` on it.

It is also worth emphasizing the limits on the length of the class and attribute names – these cannot be codified in the API, but the class name

may not be greater than `KIASClassNameMax` bytes in length, and the attribute name may not be greater than `KIASAttributeNameMax` bytes. `TBuf8` descriptors may therefore always be used, as the maximum size is always known.

Now let's look at the response we will receive:

```
NONSHARABLE_CLASS(TIASResponse) : public TBuf8<KMaxQueryStringLength>
{
public:    // But not exported
    void SetToInteger(TUint anInteger);
    void SetToCharString(const TDesC8& aCharString);
    void SetToOctetSeq(const TDesC8& aData);
    void SetToCharString(const TDesC16& aWideString);

public:
    IMPORT_C TIASResponse();

    IMPORT_C TBool IsList() const;
    IMPORT_C TInt NumItems() const;
    IMPORT_C TIASDataType Type() const;
    IMPORT_C TInt GetInteger(TInt& aResult, TInt anIndex=0) const;
    IMPORT_C TInt GetOctetSeq(TDes8& aResult, TInt anIndex=0) const;
    IMPORT_C TInt GetCharString(TDes8& aResult, TInt anIndex=0) const;
    IMPORT_C const TPtrC8 GetCharString8(TInt anIndex=0) const;
    IMPORT_C TInt GetCharString(TDes16& aResult, TInt anIndex=0) const;
    IMPORT_C const TPtrC16 GetCharString16(TInt anIndex=0) const;
};
```

A `TIASResponse` should be passed as the `aResult` parameter of `RNetDatabase::Query()`, which upon completion will contain the result of the query. As the class returns the results of the query, the 'setter' functions, while public, are internal to the stack and are not exported. Of far greater interest are the 'getter' functions. The functions `IsList()` and `NumItems()` relate to IAS's list functionality. Some operations may return more than one result, which will be encoded as a list. Unfortunately, although Symbian OS provides the ability to detect whether a list has been returned, it does not provide a way to obtain any result but the first in the resulting list. Some of the accessors will return `KErrUnsupported` if called with a non-zero list index, some will return a null value, and others will just return the first item in the list anyway. Equally, because this functionality is unused, it is not safe to assume that the `IsList()` and `NumItems()` functions work as expected. It is best to ignore list values, and only ever ask for the first entry. As all of the index parameters default to zero, they can simply be ignored.

`Type()` returns the type of result obtained from IAS. IAS can transport integers, byte sequences or user strings. It can also signal that an entry was not found in the IAS database, with a result of 'missing'. The `TIASDataType` enumeration contains the strings used to tag each of these types. If an unknown type of response is received, Symbian OS will return the type as `EIASDataMissing`.

Once the type is known the data may be extracted. There are accessors for each type of data – in the case of integers and octet sequences, these accessors are simple. Pass in a variable, and the result will be copied into it, or the function will return an error. In the case of octet sequences, the specification imposes a maximum size of 1024 bytes, but Symbian OS will not support sequences of over 125 bytes. User strings are more complicated, due to the character set. Four accessors are provided, returning either 8- or 16-bit descriptors, either as parameters passed by reference or as `TPtrC` return values. Unfortunately, defects in the implementation mean that only one of the four accessors is reliable – the one taking a `TDes8&` parameter. The 16-bit accessors will never return a descriptor, they always return `KErrNotSupported`, `KErrCorrupt`, or a null string. The 8-bit `TPtrC` accessor will work in many situations, but will return a null string if it thinks the returned string is in Unicode (UCS-2) encoding. Unfortunately, even this check is incorrect, as instead of checking the character set byte, it checks the second byte of the string. It is unlikely that this byte will be `0xff` (the Unicode character set), but if it is, or if the user string is actually in Unicode format, the result will not be as expected.

It is possible that all of these defects will be corrected in future releases. For the moment, and to maximize compatibility in future, the

Table 5.2 Return values and corresponding character set

Return value	Character set encoding	CharConv conversation value
0x00	ASCII	<code>KCharacterSetIdentifierAscii</code>
0x01	ISO-8859-1	<code>KCharacterSetIdentifier88591</code>
0x02	ISO-8859-2	<code>KCharacterSetIdentifier88592</code>
0x03	ISO-8859-3	<code>KCharacterSetIdentifier88593</code>
0x04	ISO-8859-4	<code>KCharacterSetIdentifier88594</code>
0x05	ISO-8859-5	<code>KCharacterSetIdentifier88595</code>
0x06	ISO-8859-6	<code>KCharacterSetIdentifier88596</code>
0x07	ISO-8859-7	<code>KCharacterSetIdentifier88597</code>
0x08	ISO-8859-8	<code>KCharacterSetIdentifier88598</code>
0x09	ISO-8859-9	<code>KCharacterSetIdentifier88599</code>
0xFF	UCS2	<code>KCharacterSetIdentifierUcs2</code>

only recommended accessor function is the one which takes a `TDes8&` parameter and copies the data returned into it. The return value of this variant is either the character set, in the case of a successful extraction; or an error code, in the case where the response is not a user string. There are 11 possible return values, which are listed in Table 5.2 along with the character set they represent.

The data copied into the supplied descriptor are the exact byte sequence retrieved from the remote device, so use of `CCnvCharacterSetConverter` from the `CharConv` framework is necessary to translate between the received character set and a 16-bit descriptor.

```
// assume IAS query has already been performed
_LIT(KIrResponseCharSetConversionPanic, "IRCharConv");
enum TIrCharConvPanics
{
    ECouldNotUseAsciiAsFallbackCharacterSet = 1
};
TBuf8<KMaxQueryStringLength> responseBuf;
TBuf<KMaxQueryStringLength> outputBuf;
// the two buffers above are consuming 384 bytes of stack - consider
// using heap-based buffers (HBufC or RBuf) instead
TInt charSetOrErr;
TUint charSet;

charSetOrErr = iasResponse.GetCharString(responseBuf);
if(charSetOrErr < 0)
{
    User::Leave(charSetOrErr);
}
else
{
    switch (charSetOrErr) // we know it's a char set now
    {
        case 0x00:
            charset = KCharacterSetIdentifierAscii;
            break;
        case 0x01:
            charset = KCharacterSetIdentifier88591;
            break;
        ... // and statements for the rest of the character set
        default:
            charset = KCharacterSetIdentifierAscii;
            // if unknown, assume ASCII
            break;
    }
}

CCnvCharacterSetConverter* converter = CCnvCharacterSetConverter::NewLC();

RFs fs;
User::LeaveIfError(fs.Connect());
// alternately, if in a UI thread, use CCoeEnv::Static()->FsSession() to
```

```

// retrieve shared RFs handle
CleanupClosePushL(fs);

CConvCharacterSetConvertor::TAvailability isAvailable;
isAvailable = converter.PrepareToConvertToOrFromL(charset, fs);

if (isAvailable == ENotAvailable)
{
    // then we have a problem - again just treat as ASCII
    isAvailable = converter.PrepareToConvertToOrFromL(
        KCharacterSetIdentifierAscii, fs);
    if (isAvailable == ENotAvailable) // ASCII should always be available!
    {
        User::Panic(KIrResponseCharSetConversionPanic,
            ECouldNotUseAsciiAsFallbackCharacterSet);
    }
}

TInt err;
// at this point the required conversion routine is available, or we're
// using ASCII as a fallback

err = converter.ConvertToUnicode(outputBuf, responseBuf, KStateDefault);

User::LeaveIfError(err);

// ignore any unconverted characters
// outputBuf is now ready for use

```

Example 5.1 converting between user strings retrieved from IAS and TDes16

At the end of all that we should now have a string suitable for user display!

As with octet sequences, the maximum length of a user string in Symbian OS is 125 bytes (so 62 Unicode characters), although the IAS specification allows a length of 255 bytes.

Now let's look at some example code for performing service discovery. In the first case, we're going to search for the LSAP_SEL for the default OBEX service.

```

class CIrServiceDiscoverer : public CActive
{
public:
    ...
    void DiscoverServicesL();
    virtual void RunL();
    // DoCancel would also need implementing for real applications
    ...
private:
    RSocketServ iSs;
    RNetDatabase iServices;
    TIASQuery iQuery;

```

```

    TIASResponse iResponse;
    TInt iRemoteLSAP;
    TIrdaSockAddr iDiscoveredDevice; // assume already discovered and
                                     // address stored in here
};

_LIT8(KClassDefaultObexSevice, "OBEX");
_LIT8(KAttributeLsapSel, "IrDA:TinyTP:LsapSel");

void CIrServiceDiscoverer::DiscoverServicesL()
{
    // Open a connection to socket server
    User::LeaveIfError(iSs.Open());

    // Open an IrDA service resolver
    User::LeaveIfError(iServices.Open(iSs, KIrdaAddrFamily, KIrTinyTP));

    // Run a service inquiry sequence
    query.Set(KClassDefaultObexSevice, KAttributeLsapSel,
              iDiscoveredDevice.GetRemoteDevAddr);

    services.Query(query, response, iStatus);
    SetActive();
}

void CIrServiceDiscoverer::RunL()
{
    if (iStatus.Int() == KErrNone)
    {
        switch (response.Type())
        {
            {
            case EIASDataMissing:
                User::Leave(KErrNotFound);
                break;

            case EIASDataInteger:
                // Save the remote LSAP returned
                User::LeaveIfError(result.GetInteger(iRemoteLSAP));
                break;

            case EIASDataOctetSequence:
            case EIASDataUserString:
            default:
                // We asked for an LSAP, and received something other than an
                // integer. There's something going wrong...
                User::Leave(KErrCorrupt);
                break;
            }
        }
    }
    else
    {
        // Service discovery failed.
        User::Leave(err);
    }
}

```

Example 5.2 Service discovery code, integer type

In this next case, we're going to demonstrate retrieving the remote device name, by querying for the general 'Device' class, with an attribute of 'DeviceName'. These are standard names and attributes from the IrDA specification that allow us to retrieve the remote device's nickname. We'll use the same class names in Example 5.3 as we did in Example 5.2, but obviously these implement different functionality, so you'd need to combine the two in some way if you want both functions!

```
class CIrServiceDiscoverer : public CActive
{
public:
    ...
    void DiscoverRemoteDeviceNameL();
    virtual void RunL();
    // DoCancel would also need implementing for real applications
    ...
private:
    RSocketServ iSs;
    RNetDatabase iServices;
    TIASQuery iQuery;
    TIASResponse iResponse;
    TBuf16<KMaxQueryStringLength> iDeviceName;
    TIrdaSockAddr iDiscoveredDevice; // assume already discovered and
                                    // address stored in here
};

_LIT8(KClassDevice, "Device");
_LIT8(KAttributeDeviceName, "DeviceName");

void CIrServiceDiscoverer::DiscoverRemoteDeviceNameL()
{
    // Open a connection to socket server
    User::LeaveIfError(iSs.Open());

    // Open an IrDA service resolver
    User::LeaveIfError(iServices.Open(iSs, KIrdAddrFamily, KIrtTinyTP));

    // Run a service inquiry sequence
    query.Set(KClassDevice, KAttributeDeviceName,
        iDiscoveredDevice.GetRemoteDevAddr());

    iServices.Query(query, response, iStatus);
    SetActive();
}

void CIrServiceDiscoverer::RunL()
{
    if (iStatus.Int() == KErrNone)
    {
        switch (response.Type())
        {
            case EIASDataMissing:
                User::Leave(KErrNotFound);
                break;
        }
    }
}
```



```

case EIASDataUserString:
{
    // Save the string returned
    TBuf8<KMaxQueryStringLength> rawBytes;
    TInt charSet = result.GetUserString(rawBytes);
    if (charSet < 0)
    {
        User::Leave(charSet);
    }
    ConvertCharacterSetL(charSet, rawBytes, iDeviceName);
    break;
}
case EIASDataInteger:
case EIASDataOctetSequence:
default:
    // We asked for a device name, and received something other
    // than a string. There's something going wrong...
    User::Leave(KErrCorrupt);
    break;
}
}
else
{
    // Service discovery failed.
    User::Leave(iStatus.Int());
}
}
}

```

Example 5.3 Service discovery code, user string type

In this example, the conversion from raw bytes in the supplied character set to Unicode half-words has been left to a function `ConvertCharacterSetL`, which would use most of the code from Example 5.1, but obviously without the buffers that were declared in that example, as this function supplies them.

5.3.6 Service Registration

Service registration is performed via `TIASDatabaseEntry`.

```

NONSHARABLE_CLASS (TIASDatabaseEntry) :
{
    public TPckgBuf<TIASDatabaseEntryV001>
public:
    IMPORT_C void SetClassName(const TDesC8& aClassName);

    IMPORT_C void SetAttributeName(const TDesC8& aAttributeName);

    IMPORT_C void SetToInteger(const TUint aInteger);
    IMPORT_C void SetToCharString(const TDesC8& aCharString);
    IMPORT_C void SetToCharString(const TDesC16& aWideString);
    IMPORT_C void SetToOctetSeq(const TDesC8& aData);
};

```

`SetClassName()` and `SetAttributeName()` set the entries class and attribute fields respectively, whilst the remaining methods set the data contained within the entry. Of the four accessors, the three descriptor-related methods have caveats attached, due to descriptor length limitations. As mentioned before, Symbian OS limits the size of octet sequences and user strings to 125 bytes. Calling the setter methods with a larger buffer will result in a descriptor overflow panic, unless the buffer size is greater than 256 bytes. If the buffer is this large, the stack will raise panic `Irda Panic-1` instead.

Another point to note is the character-encoding scheme used by the two `SetToCharString()` methods. These methods set user strings, using either ASCII or UCS-2 encoding. There is no way to specify a different encoding method.

Once the `TIASDatabaseEntry` object has been populated, it may then be passed to `RNetDatabase::Add()` to be stored in the IAS database.

```
class CIrRegisterLocalService : public CActive
{
public:
    ...
    void RegisterServiceL();
    virtual void RunL();
    ...
private:
    RSocketServ iSs;
    RNetDatabase iServices;
    TIASDatabaseEntry iRecord;
    Tint iObexPort; // should get this when opening listening socket and
                   // calling AutoBind(), ie. socket must be open before
                   // we attempt to register the service!

};

_LIT(KClassDefaultObexService, "OBEX");
_LIT(KAttributeLsapSel, "IrDA:TinyTP:LsapSel");

void CIrRegisterLocalService::RegisterServiceL()
{
    User::LeaveIfError(ss.Open());

    // Open an IrDA service resolver
    User::LeaveIfError(services.Open(ss, KIrdaAddrFamily, KIrTinyTP));

    // Build an IAS record
    record.SetClassName(KClassDefaultObexService);
    record.SetAttributeName(KAttributeLsapSel);
    record.SetToInteger(iObexPort);

    // Insert record into database
    services.Add(record, iStatus);
    SetActive();
}
```

```
void CIrRegisterLocalService::RunL()
{
    User::LeaveIfError(status.Int()); // service failed to register
}
```

Example 5.4 Inserting a record in the local IAS database

One possible cause of errors with this code is re-registering an entry that already exists. In this case the IAS server will detect the conflict and return a `KErrAlreadyExists` value.

It is also worth noting that closing the `RNetDatabase` handle does not remove any entries registered through this handle – see the section ‘5.3.7 Service removal’ for details of how to deregister a service.

Hint bits

In addition to the IAS database, IrDA also provides a small set of device hint bits, which are returned at the discovery stage. This allows a simple filtering operation to be performed before deciding to connect to a device.

The Symbian API to retrieve or set these bits is through socket options on an open `RSocket` instance. In this example, we’ll enable the hint bit that indicates that we provide OBEX services.

```
// iSocket is an already opened socket that will be used for the service
// we are providing

void SetHintBitL()
{
    // Retrieve the current hint byte (OBEX is in the second byte)
    TDes8<sizeof(TUint)> buffer;
    User::LeaveIfError(iSocket.GetOpt(KSecondHintByteOpt, KLevelIrLap,
        buffer));
    TUint8 secondByte = buffer[0];

    // Force on the OBEX bit
    secondByte |= KIrObexMask;
    buffer[0] = secondByte;

    // Set new hint byte
    User::LeaveIfError(iSocket.SetOpt(KSecondHintByteOpt, KLevelIrLap,
        buffer));
}

// Run an OBEX server
...
// OBEX server terminated

void ClearHintBitL()
{
    // Retrieve the current hint byte
    User::LeaveIfError(socket.GetOpt(KSecondHintByteOpt, KLevelIrLap,
        buffer));
```

```

TUint8 secondByte = buffer[0];

// Force the OBEX bit off
secondByte &= ~KIrObexMask;
buffer[0] = secondByte;

// Set the new hint byte (clearing the bit)
User::LeaveIfError(socket.SetOpt(KSecondHintByteOpt, KLevelIrLap,
                                buffer));
}

```

Example 5.5 Setting and clearing hint bits on the local device

This example shows both setting and clearing a hint bit and shows an important aspect of the API – the stack does not manage hint bits for the programmer. As a result, setting a hint bit without first getting its current value may lose hint bits set by other applications. This does lead to potential race conditions with two applications trying to add a new hint bit, where both applications could read the current hint bits, OR in their new hint bit, then write the new value to the stack. In that case, unless they were both setting the same bit, one of the writes would be lost.

However, the biggest problem is seen when clearing hint bits, as there is no way to be sure that another application is not providing a service that requires the same hint bit to be set as your application. The consequence of this is that we recommend that you do not ever clear hint bits – that way you do not risk breaking other application’s use of them. Since they are *hint* bits, indicating roughly which services a device supports, this should not result in any major problems – other devices may connect unnecessarily because they believe the local device is offering a service which it is not, but this can happen in normal operation, as one hint bit covers a series of possible services.

5.3.7 Service Removal

When removing an IAS record, you need a `TIASDatabaseEntry` with the same details as the one you used to register your service. This can be the same instance you used to insert the record into the database, if you kept it around, or it can be an entirely new object created using the same parameters.

Once the class and attribute names have been set, the object is passed to `RNetDatabase::Remove()`, and the entry will be removed from the database or an error raised if the record cannot be found. There is no need to set the attribute value.

It is also worth noting that the `RNetDatabase` object may be a completely unrelated connection to the one which originally registered the entry – it may even be in a different process.

```

class CIrDeregisterLocalService : public CActive
{
public:
    ...
    void DeregisterServiceL();
    virtual void RunL();
    ...
private:
    RSocketServ iSs;
    RNetDatabase iServices;
    TIASDatabaseEntry iRecord;
};

_LIT(KClassDefaultObexService, "OBEX");
_LIT(KAttributeLsapSel, "IrDA:TinyTP:LsapSel");

void CIrDeregisterLocalService::DeregisterServiceL()
{
    User::LeaveIfError(iSs.Open());

    // Open an IrDA service resolver
    User::LeaveIfError(iServices.Open(ss, KIrdaAddrFamily, KIrTinyTP));

    // Build an IAS record
    iRecord.SetClassName(KClassDefaultObexService);
    iRecord.SetAttributeName(KAttributeLsapSel);

    // Remove record from database
    iServices.Remove(iRecord, iStatus);
    SetActive();
}

void CIrDeregisterLocalService::RunL()
{
    User::LeaveIfError(status.Int()); // service was not deregistered
}

```

Example 5.6 Deregistering a service from the local IAS database

5.3.8 Client Sockets

Now that we've seen how to query for devices, register local service and query for remote services, and set the appropriate hint bits, it's time to look at actual data transfer. First we'll consider outbound connections in this section, then incoming connections in 'Server sockets'.

Connecting a client socket to a server running on a remote machine is quite simple. First we need to open the socket. The `RSocket::Open()` call takes a number of parameters – protocol family, socket type and protocol identifier. Constants are provided for all of these, but in the case of IrDA, the most important two are the protocol family (`KIrdaAddrFamily`), and the socket type (defined in `es_sock.h`). It is still worth using the appropriate protocol identifier, however.

Table 5.3 summarizes the correct parameters for the call to `Open`:

Table 5.3 Call parameters for `Open`

	Address family	Socket type	Protocol ID
IrMUX	KIrdaAddrFamily	KSockDatagram	KIrmux
TinyTP	KIrdaAddrFamily	KSockSeqPacket	KIrTinyTP

Next, the remote address and port are found, using the techniques for device and service discovery that we covered earlier in the chapter, and passed to the stack by means of a call `Connect()`. When the `Connect()` returns, a link will have been established or an error will be signalled.

```
CIrOutboundConnection : public CActive
{
    enum TState
    {
        EDiscoveringDeviceAndService,
        EConnectingSocket
    };
public:
    ...
    void ConnectSocketL();
    virtual void RunL();
    ...
private:
    RSocketServ iSs;
    RSocket iSocket;
    TIrdaSockAddr iAddr;
    TUint8 iPort;
    TState iState;
};

void CIrOutboundConnection::ConnectSocketL()
{
    {
        iState = EDiscoveringDeviceAndService;
        User::LeaveIfError(ss.Open());

        // Open an IrDA socket
        User::LeaveIfError(socket.Open(ss, KIrdaAddrFamily, KSockSeqPacket,
                                     KIrTinyTP));

        // Run device discovery, connect to IAS, and find port.
        // Return port and device addresses in the supplied parameters.
        DiscoverDeviceAndService(iAddr, iPort, iStatus);
        SetActive();
    }
}

void CIrOutboundConnection::RunL()
{
    {
        if (iStatus.Int() == KErrNone)
        {

```

```

switch (iState)
{
case EDiscoveringDeviceAndService:
    iState = EConnectingSocket;
    // Build a complete TIrdaSockAddr from discovery
    // results. We could have made DiscoverDevice... function
    // return the address and port precombined, but chose not
    // to illustrate the two parts of the address
    iAddr.SetRemotePort(iPort);

    // Open connection
    socket.Connect(iAddr, iStatus);
    SetActive();
    return;

case EConnectingSocket:
    // Now connected to remote device.
    break;
}
}
else
{
    User::LeaveIfError(status.Int());
}
}

```

Example 5.7 Connecting to a remote device

DiscoverDeviceAndService() is not shown, but is a combination of the device discovery and service search functionality we've already seen.

Now we've seen how to connect an outbound socket, we'll look at creating inbound (a.k.a. server) sockets.

5.3.9 Server Sockets

Running an IrDA server on Symbian OS requires a 'server' or 'listening' socket. This socket provides an endpoint which a remote device may connect to, and so exchange data.

As with all listening sockets, the sequence is: open the socket, bind to the desired local port, listen for connections, and finally issue an asynchronous Accept() call. The general framework has already been covered in chapter 3, so this section will only deal with the IrDA specializations.

Opening a socket is exactly the same as for outgoing connections, and uses the same parameters. Once the socket has been opened, the next step is binding to a local LSAP_SEL. Due to the limited range of values available, it is more likely that a port will already be in use if hard-coded (or 'well-known') values are used. As IrDA provides the IAS database to discover channel identifiers, it is strongly recommended that you allow the stack to select an appropriate LSAP_SEL for your application. An invalid LSAP_SEL constant is provided in `ir_sock.h` for this purpose – binding to `KAutoBindLSAP` will lead to the stack selecting the

lowest numbered unused LSAP_SEL. A call to `RSocket::LocalName()` will retrieve the selected port, after which point you can register your service in the IAS database, as described previously in '5.3.6 Service registration'.

As a final point to note when dealing with listening sockets, accept queue sizes greater than one have not been extensively tested. Whilst the IrDA protocols and the Symbian stack should handle multiple simultaneous connections to the same local LSAP_SEL, for maximum reliability it is recommended that this is not relied upon.

```
class CIrInboundConnection : public CActive
{
    enum TState
    {
        ERegisteringService,
        EListening
    };
public:
    ...
    void StartListeningL();
    virtual void RunL();
    ...
private:
    RSocketServ iSs;
    RSocket iListener;
    RSocket iAcceptor;
    TUint8 iPort;
    TState iState;
};

void CIrInboundConnection::StartListeningL()
{
    iState = ERegisteringService;
    User::LeaveIfError(ss.Open());

    // Open an IrDA socket
    User::LeaveIfError(iListener.Open(ss, KIrdaAddrFamily, KSockSeqPacket,
        KIrTinyTP));

    // Create a TIrdaSockAddr
    TIrdaSockAddr addr;
    addr.SetLocalPort(KAutoBindLSAP);

    // Bind to port
    User::LeaveIfError(iListener.Bind(addr));

    // Start listening
    User::LeaveIfError(iListener.Listen(1));

    // Register service in IAS
    User::LeaveIfError(iListener.LocalName(addr));
    iPort = addr.GetHomePort();
    RegisterThisService(iPort, iStatus);
    SetActive();
}
```



```

void CIrInboundConnection::RunL()
{
    if (iStatus.Int() == KErrNone)
    {
        switch (iState)
        {
            case ERegisteringService:
                iState = EListening;
                // Wait for a connection
                listener.Accept(iAccepting, iStatus);
                SetActive();
                return;

            case EListening:
                // do something with newly connected socket!
                // eg. iSomeInterface.NewConnection(iAccepting);

                // Wait for the next connection
                listener.Accept(iAccepting, iStatus);
                SetActive();
            }
        }
    }
    else
    {
        User::Leave(iStatus.Int());
    }
}

```

Example 5.8 Listening for an incoming connection

Again, we have not shown `RegisterThisService()` but it can be easily created based on the earlier examples. Neither have we shown where connected sockets are used, beyond the comment about `iSomeInterface` in the comment on the `EListening` case, as this is also application-specific detail.

5.3.10 Reading and Writing

Reading and writing data over IrDA sockets is performed exactly as with any other datagram-based sockets. The major point to note is that even TinyTP only exposes a datagram interface – there is no stream support. As a result of this, datagram boundaries are respected, and so whole datagram reads must normally be queued. However, ESOCK does have a system known as datagram read continuation, which allows partial reads on datagram protocols. This is achieved by passing `KSockReadContinuation` into the `Recv()` call in the flags parameter – see chapter 3 for full details.

Two socket options are also useful here: `KHostMaxDataSizeOpt` and `KRemoteMaxDataSizeOpt` allow discovery of the packet size in use on the link. Incoming datagrams will never be larger than this, so reads of `KHostMaxDataSizeOpt` size will never lose data, or require the use of read continuations. Some devices limit the packet size – the IrDA protocols allow for a range of sizes up to 2048 bytes.

Similarly to reads, writes must also respect datagram boundaries. In the case of writes, this means not asking the stack to send more data than `KRemoteMaxDataSizeOpt`. This means that fragmentation of large packets must be performed by the application, as the stack will refuse to send such datagrams.

There is one situation where larger packets than the transport packet size may be sent – when using TinyTP segmentation and reassembly. This is a little-used feature of TinyTP which allows a single datagram to be segmented across a number of LM-MUX datagrams, and then reassembled on reception. The reason that it is rarely used is that a number of implementations have contained defects, so causing interoperability problems. A Symbian OS socket option, `KTinyTPDisabledSegmentation`, exists to force the stack to ignore requests to reassemble datagrams – this may help when running against stacks which ignore the maximum buffer size for reassembly advertised by the host. The data received is still valid, apart for the additional datagram boundaries, however this can cause serious problems if your application relies on the framing of TinyTP to provide a datagram interface as you will no longer be able to determine where the ‘real’ datagram boundaries are. It is recommended that you always set this socket option unless you need the framing provided by TinyTP.

5.3.11 Socket Options

These options may be passed to `RSocket::SetOpt()`, in order to prompt the stack to perform a variety of actions. Unless otherwise indicated, all of these calls require the `LocalServices` capability.

```
KUnexpeditedDataOpt
KExpeditedDataOpt
```

Use expedited data packets to carry the data. These data packets are transferred in IrLAP UI frames, which exempts them from reliable delivery. They are also transmitted in preference to non-expedited packets, and on the receiver side will short-circuit queues. As a result, using these packets breaks in-order delivery. There is no way to know that a particular packet received by a remote device was transmitted as an expedited packet, and no requirement on a receiver to agree to handle such packets.

Unlike the other socket options, the two options above may only be used on individual socket writes – rather than in a call to `SetOpt()`.

```
KDiscoverySlotsOpt
```

Sets the number of discovery slots to use when running a discovery sequence. A single slot discovery will return most quickly, but may fail to return any devices when more than one device responds to the query. Valid values are 1, 6, 8 or 16. The recommended default value is 6.

```
KUserBaudOpt  
KHostMaxDataSizeOpt  
KRemoteMaxDataSizeOpt  
KHostMaxTATimeOpt  
KIrIrLapDisableResetOpt
```

These values should not generally be changed, but can be read to determine the status of an established link. Under certain situations it may be necessary to change the host values, but as this is strongly discouraged, this use is not discussed here. Generally, the highest baud rate supported by both stations is used, together with the highest maximum turnaround time. This combination provides maximum throughput, although at the cost of a slightly increased maximum latency. `KUserBaudOpt`, `KHostMaxDataSizeOpt` and `KMaxTATimeOpt` all need the `NetworkControl` capability.

```
KLocalBusyDetectedOpt  
KLocalBusyClearedOpt
```

These flags allow the programmer to set or clear a 'local busy state'. This state acts as flow control on the lowest level of the link – in a busy state, any packets sent by the remote station are rejected but the link is maintained until the busy state is cleared. It is recommended not to use this option, especially as `KLocalBusyDetected` needs the `NetworkControl` capability.

```
KDiscoveryResponseDisableOpt
```

Disable responses to incoming discovery attempts for three seconds. This allows the local station to establish itself as the primary, without other stations connecting in. This should not generally be required, but can increase connection reliability when interoperating with certain other IrDA implementations. As such it is recommended that it be set if the application has the `NetworkControl` capability. Any failure to set this, however, should not be regarded as critical.

```
KFirstHintByteOpt  
KSecondHintByteOpt
```

Set or get the first and second hint bytes. These bytes provide coarse-grained service discovery, and are described in section 5.3.6. However, setting them requires the `NetworkControl` capability.

```
KTinyTPLocalSegSizeOpt  
KTinyTPRemoteSegSizeOp
```

Find or set the local and remote TinyTP segment size. This value is used when TinyTP Segmentation and Reassembly is in use. Under normal circumstances, due to known problems in other implementations, the IrDA SIG recommends that SAR not be used. When setting the local size, values greater than 2048 bytes may not be used.

```
KTinyTPDisabledSegmentation
```

Despite advertising a zero length reassembly buffer, some TinyTP implementations may still attempt to send SAR data. Setting this flag will tell the socket to treat individual TinyTP PDUs as individual SDUs, even when the header data claims that multiple packets should be concatenated together. Apart for providing more SDU boundaries than the sender intended, the data are unaffected.

5.3.12 Ioctl

These ioctls may be passed to `RSocket::Ioctl()`, in order to prompt the stack to perform a variety of actions. Unless otherwise indicated, all of these ioctls require the `LocalServices` capability.

Ioctls differ from socket options in that they are asynchronous – as a result many of these are callback notifications.

```
KDiscoveryIndicationIoctl
```

Completes when a remote station discovers the local machine. The provided descriptor is filled with the `TNameRecord` related to the device which has discovered the local machine.

```
KExclusiveModeIoctl  
KMultiplexModeIoctl
```

Requests the stack treat the current socket as ‘exclusive’. This prevents all other sockets from sending data, so preventing deadlocks between multiple IrMUX level sockets. This ioctl will fail unless the socket specified is the only socket currently open on the link. Trying to set exclusive mode also requires the `NetworkControl` capability.

```
KIrmuxStatusRequestIoctl  
KIrlapStatusRequestIoctl  
KIrlapStatusIndicationIoctl  
KIrmuxStatusIndicationIoctl
```

These ioctls are in fact largely identical, and report the major status of the IrLAP link. The only difference is that the `Request` ioctls report the current status, while the `Indication` ioctls wait for a state change.

```
KIdleRequestIoctl
```

KIdleClearRequestIoctl

These ioctls may be used to allow a link to be placed into exclusive mode even when multiple open sockets exist. Only if all other open sockets have been marked as idle will the exclusive mode transition is allowed.

KDisconnectIndicationIoctl

Completes to indicate that the socket has been closed. This will be followed by any outstanding read or write attempts on the socket completing with an error. The supplied descriptor is filled with an integer representing the reason for the link’s disconnection (see Table 5.4).

Table 5.4 Reason for link disconnection

Value	Reason
0x01	User request
0x02	Unexpected IrLAP disconnect
0x03	Failed to establish IrLAP connection
0x04	IrLAP reset
0x05	Link management initiated disconnect
0x06	Data delivered on disconnected LSAP connection
0x07	Non-responsive LM-MUX client
0x08	No available LM-MUX client
0x09	Connection half open
0x0A	Illegal source address
0xFF	Unspecified disconnect reason

KIrlapResetRequestIoctl

Requests that the underlying IrLAP link be reset. This will discard any data currently stored in the protocol’s buffers, on both sides of the link. An unknown amount of data will therefore be lost. This requires the NetworkControl capability.

KIrlapResetIndicationIoctl

Completes to indicate that the underlying IrLAP link has been reset.

```
KIrlapDisconnectRequestIoctl
```

Requests that the underlying IrLAP link be disconnected. This requires the `NetworkControl` capability.

5.3.13 Monitoring Link Status

To monitor status changes on the link, a `Publish` and `Subscribe` key has been defined. This key progresses through seven states, showing the stack being loaded or unloaded, as well as discovering a remote device, losing the discovered device, connected, disconnected, and the link being interrupted.

A typical sequence might look like:

```
TIrdaStatusCodes::EIrLoaded
TIrdaStatusCodes::EIrDiscoveredPeer
TIrdaStatusCodes::EIrLostPeer
TIrdaStatusCodes::EIrConnected
TIrdaStatusCodes::EIrBlocked
TIrdaStatusCodes::EIrDisconnected
TIrdaStatusCodes::EIrUnloaded
```

The `Publish` and `Subscribe` key is in category `KIrdaPropertyCategory` (defined as being `KUidSystemCategoryValue`), and the key itself is `KIrdaStatus`.

Watching for `EIrDiscoveredPeer` is another method of waiting for remote discoveries, therefore avoiding the need to use the `ioctl` value `KDiscoveryIndicationIoctl`. It is important to note, however, that `Publish` and `Subscribe` is not intended as a reliable service – in the sense that if values are published very rapidly then some listeners may not receive all published values. As such, if it is important that you are notified on remote discoveries then using the `ioctl` may be the preferable option.

5.4 Summary

In this chapter we have learnt:

- about the general concepts in IrDA
- how to discover other devices using IrDA
- how to query the IAS (service) database on remote devices
- how to add entries to our local IAS database to allow other devices to connect to services that we offer
- how to connect outgoing and receive incoming socket connections
- about the socket options that IrDA offers for additional control of the stack.

6

IP and Related Technologies

IP is the most common protocol for data communications today, and is likely to become even more popular in the near future as voice services migrate to use IP. Most mobile devices have IP connectivity for accessing the Internet, connecting to email services, and for using other services that network operators make available over GPRS. Additionally, increasing numbers of mobile devices now enable IP over more than just the GPRS network; both WLAN and the Bluetooth PAN profile enable connectivity to a wider range of services, including ad-hoc local networks (such as multi-player games) and convergent networks (such as UMA and IMS) where the same services can be accessed over different technologies, in addition to being used for access to corporate networks in the workplace.

Symbian OS is an open operating system, so by making IP protocols available to application developers, a large number of new applications and services can be implemented. Users have the ability to download and install new applications as they desire (possibly using IP for the download as well!), and operators and phone manufacturers can differentiate their devices by creating new IP-based services and applications for their subscribers.

The purpose of this chapter is to enable you to implement IP-based applications, and make full use of the rich set of APIs available on Symbian OS. We also provide guidance on ensuring that the applications work well over different bearer technologies, such as GPRS and WLAN. This chapter assumes that you have some prior knowledge or experience in using IP protocols, and are familiar with the sockets interface as described in Chapter 3.

6.1 IP Networks Overview

6.1.1 An Overview of GPRS

Before we start on the Symbian OS-specific details, we'll give a brief overview of GPRS networks, as they may not be familiar to many developers. As with most network technologies that carry IP, there are some configuration steps that are specific to GPRS, so we'll explain the key parameters that need to be configured. GPRS networks also offer some features which differ from other technologies, such as secondary contexts – which provide quality of service (QoS) for traffic at the link layer – we'll discuss these as well.

One thing worth noting is that when we talk about GPRS in this chapter, we mean GPRS services over both the GERAN (also known as a 2.5G network) and UTRAN (also known as a 3G network).

Let's look at how a GPRS connection is created to see the different parts of the system and where the configuration parameters come from. We'll ignore some of the really low level details, like the physical channel over which these commands are transported, and the parts of the network that are transparent as far as we're concerned (like the actual base station that receives the radio signals), and just look at the higher level connection process.

The two key items in Figure 6.1 are the *serving GPRS support node* (SGSN) and the *general GPRS support node* (GGSN). The GGSN is the really interesting node from the developer point of view, as this is connected to the network we wish to access – typically either an operator network, or the Internet.

The first item of configuration information we need is the *access point name* (APN), which is passed to the SGSN by the mobile handset. This is then used by the SGSN to discover which GGSN we wish to connect to. Different GGSNs are connected to different external networks, so, for example, one GGSN may provide access to the Internet, another may provide access to the operator's walled garden network.

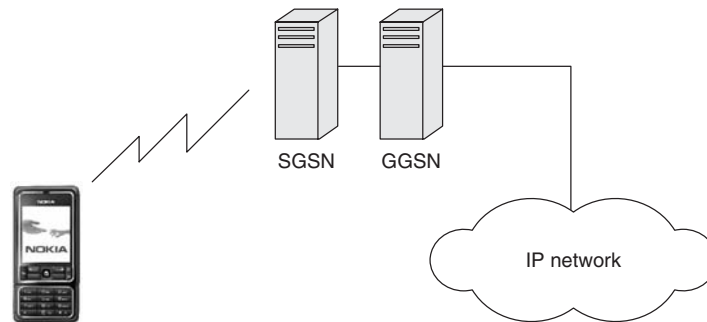


Figure 6.1 Component parts of a GPRS connection

The other commonly required items of configuration information is a username and password, although these are usually set to well known values used by all users of a particular service.

There are various sites on the Internet that list the APNs, usernames and passwords for popular mobile operators; or settings can be sent over-the-air in a specially formatted SMS containing configuration information – these can typically be requested using the appropriate operator's website. Some recent devices also have a very useful function where they identify the operator from your SIM card and populate the access point settings from a local database containing access point information for many network operators.

The configuration information is used to activate a network connection known as a *PDP context*, or often just as a 'context'. This is a connection from the handset to the GGSN, and provides access to the network attached to the GGSN.

Why so many APNs?

A common question is why, when all the PDP contexts we are using support IP, do we need more than one access point? Surely all traffic could be sent over the same IP link to a variety of different destinations? The common reason for using multiple APNs (and therefore multiple PDP contexts) is for billing purposes. It turns out to be a lot easier to build a network if you can just record the amount of data transferred through one of the GPRS support nodes (S/GGSN) rather than having to examine every IP packet and determine which type of service (eg. internet, MMS, operator walled garden network) it relates to, and then run multiple counters for the different service types¹. Similarly, it is easier to grant some users, perhaps depending the network tariff to which they are subscribed, access to a greater or lesser set of services by allowing or blocking their use of certain APNs; for example, in the UK some network operators do not provide general Internet access to prepaid customers. Most networks support between one and three access points, although obviously this is highly variable.

Symbian OS, when using a cellular modem with appropriate capabilities, can support multiple contexts to multiple different APNs – although often the number is limited because either the cellular modem or the network have a per device limit on the number of contexts they can or will support. There is also a maximum number of supported contexts defined in the 3GPP specification, although we have yet to observe a case where this is the limiting factor.

¹ Given the increased amount of processing power available today this is more feasible than it once was, but still adds extra overhead to the per-packet processing.

Quality of service on GPRS networks

A feature that is unique to GPRS in implementation, if not in concept, is the *secondary context*. Unlike the PDP contexts we referred to before, which would more accurately be referred to as *primary contexts*, a secondary context carries some subset of traffic on the IP link. It is used to provide differing quality of service (QoS) to different connections, typically different TCP or UDP connections. Thus a device would have a single primary context and zero or more secondary contexts active, all using the same IP address and connected to the same GGSN. Each of the secondary contexts would typically have a non-default QoS setting – for example, it might provide a lower latency and lower jitter service for real-time voice services, or a high bandwidth connection for video streaming from a server.

Whilst packets leaving the local device can easily be transmitted on the appropriate primary or secondary context, it is necessary to provide additional information to the GGSN to allow it to map data coming from the external network onto the correct context. This is achieved through the use of a traffic flow template (TFT). This is simply a data structure that allows the local device to specify certain characteristics that can be used to assign packets arriving at the GGSN from the external network to the correct context. TFTs can provide a range of parameters to classify traffic, for example TCP or UDP port number, or IPSec SPI.

When using the `RSubConnection` API in Symbian OS, covered in section 6.6, the creation and handling of TFTs is taken care of by the GPRS QoS implementation so the application need only specify its QoS requirements through the API.

In addition to QoS on secondary contexts, it is also possible for primary contexts to have specialised QoS parameters. Whilst not used for the majority of applications and access points, this feature is sometimes used to provide QoS-enabled links on networks where secondary contexts are not supported.

Limitations on GPRS networks

As discussed earlier in this chapter, some networks have a limitation on the number of active PDP contexts they support. The maximum number varies between networks, and even on a single network it changes occasionally as network operators decide to increase the number of active contexts allowed. It is possible, however, to see context activation fail because there are too many contexts already active. In this case it is often best to indicate the reason for failure to the user (in less technical terms, obviously) and let them solve the problem, as any attempt to shutdown other connections can be equally disruptive to the user as having an application fail to create a connection. Both S60 and UIQ have connection manager applications that allow users to monitor and

close active connections – however, many users are unlikely to be able to make a sensible decision about which connection to close, so this feature is only really useful to the more technical users.

6.1.2 An Overview of Multihoming

Following on logically from what we've just seen about GPRS with different services being offered over different APNs, it is obvious that if we want to support multiple services spanning multiple APNs that we need the ability to use several network connections at the same time. Of course the connection lifetimes don't need to be the same – network connections can start and stop independently, but it should be possible to create a new network connection even if there is an existing connection. Figure 6.2 shows a typical scenario.

One common problem when using multiple connections, which is often seen on PCs, is the routing ambiguity it creates. In other words, it is not obvious which packets should be sent through which interface, or which interface's IP address should be used as the source address for new connections. The simple solution to this is to designate one interface as the default – however, if different services are available over the different interfaces (eg. MMS services are typically only available over a connection to the network operator's MMS network) then the problem becomes much harder to solve. In some cases, when using the traditional approach of a single routing table to decide which interface to send traffic over, services may just stop working if their traffic is sent over the wrong interface. What is more, it often takes someone very familiar with IP networking to sort the problem out – and this isn't something you expect from the vast majority of phone users.

In cases where a device is connected to multiple networks that are administered by different organisations the problem can get even worse. Many networks use 'private' IP addresses, as defined in RFC1918. As there is no central control over their allocation, it is perfectly feasible to be using one network, for example WLAN, and have an IP address of 192.168.1.1, and also be connected to your network operator's MMS network where you could be given exactly the same IP address! At this point an IP stack using a traditional routing table approach would be completely stuck. Even when the same address is not given to both local interfaces, there are still problems with deciding where packets should go – which interface do you use for a packet destined for 192.168.1.5, for example? There's no way to tell, as the routing table is likely to offer both interfaces as equally good routes – even though the device to which you want to connect is reachable on only one of the two networks.

Symbian OS solves these multihoming problems by getting applications to declare which interface they want to use, through use of the `RConnection` API, then allowing them to associate their sockets

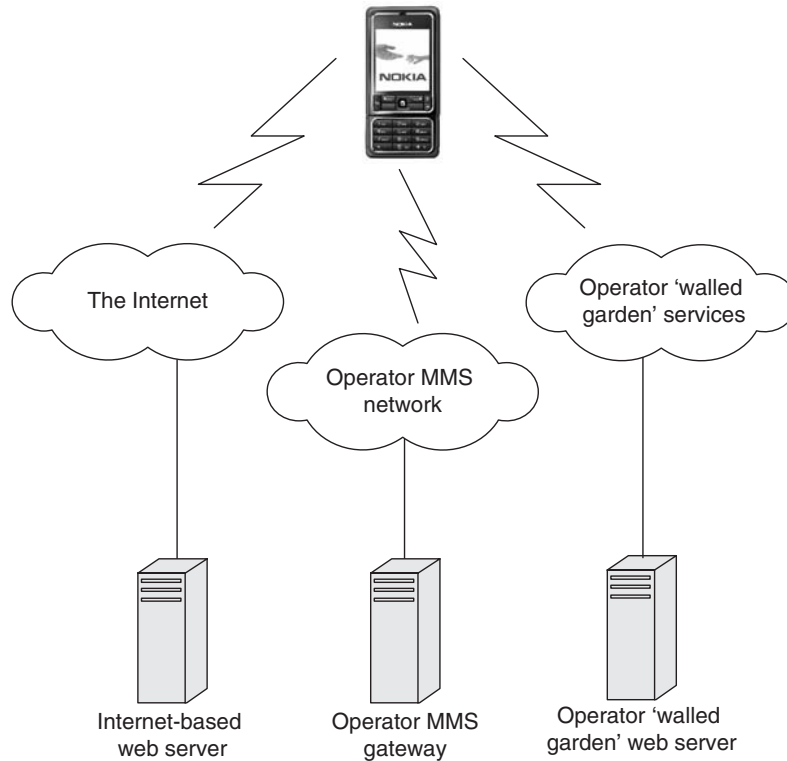


Figure 6.2 A typical multihoming scenario

with that connection. Host resolver connections can also be scoped in the same way. This means that an application is unaware of other, potentially conflicting, network connections that are active on the device, and therefore works as it would if there were only the one connection.

6.2 IP Networks and Symbian OS

An IP network for a mobile phone can provide many services over a number of different bearers:

- Network operators provide IP-based services on their own networks, for example, MMS, music and video downloads.
- Most network operators offer access to the Internet as well as their own walled garden services.
- Ad-hoc networks can be created using local area networking technologies such as Bluetooth, for example, for multi-player gaming.

- Service providers on the internet may provide their own IP services, for example, Skype.

Because of the variety of networks available, you need to give careful consideration to which of these different networks are most suitable for their application. Symbian OS aims to make it as easy as possible to support any or all of these network technologies by providing common interfaces that enable applications to use different bearers and network types without being concerned with the details of a specific network technology or connection.

It is important to remember that devices running Symbian OS are 'mobile'. This means availability of networks may change as the device moves geographically – this can occur over large distances, such as going to a different country where a different service provider may be used; and over shorter distances, such as travelling on the train, where the availability of the GPRS network can vary from minute to minute. Applications need to be able cope with this dynamic behavior. This is an important point to consider when developing for mobile devices, particularly if you are porting an application from another platform which has been designed to use a fixed network with a very high availability (as would be the case for many PC-based applications, for example).

This chapter discusses the basics of how to create a network connection, how to use it to enable IP-based services on Symbian OS and what to consider when creating applications for mobile devices.

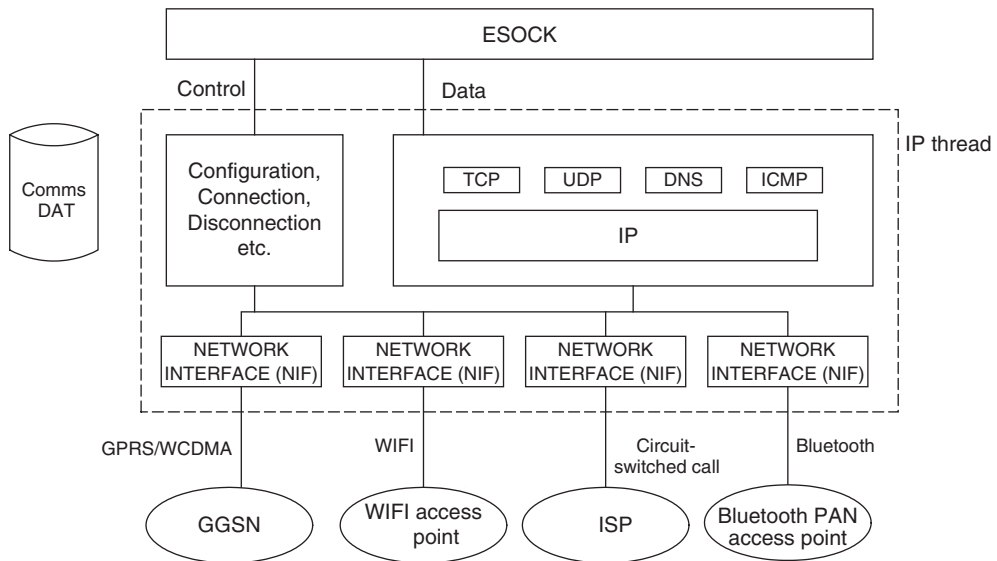


Figure 6.3 Symbian OS networking subsystem and bearer technologies

6.2.1 Getting Started

One of the most important concepts to understand for IP-based applications on Symbian OS is the *Internet Access Point* (IAP). An IAP is a collection of settings that define how a connection to a particular network is made. These settings are stored in the communications database, *CommsDat* (discussed in section 6.3.2) – the application itself does not need to know the parameters required to connect to different networks. It can use the system-wide connection preferences, or the application can provide a specific connection preference.

Here are some examples of information stored relating to a specific IAP:

- The phone number, for a dial-up ISP.
- The access point name (APN), for a GPRS connection.
- The SSID of a WLAN access point.

These details will be discussed further in section 6.3.2, but the concept of an IAP is important to understand as it is used throughout the rest of the chapter. An IAP can also be thought of as representing a single network interface, and each IAP refers to a single interface.

CommsDat is a new component in Symbian OS v9.1, and replaces CommDB which performed the same role in earlier releases. A compatibility layer exists for clients of CommDB using the C++ API, however there is no longer any support for querying the comms database using SQL statements.

The three most important classes for IP-based applications on Symbian OS are `RConnection`, `RSocket` and `RHostResolver`. These have already been introduced in Chapter 3 but are discussed with particular reference to IP here. `RConnection` was first introduced in Symbian OS v7.0s in order to support multihoming. `RConnection` allows an application to select which IAP to use for its network connection, and to associate its sockets with that connection. This is very important because this ensures that the IP traffic from those sockets is sent to the correct network. `RConnection` also has APIs for gathering information about a network connection such as the status, connection information etc.

The `RConnection` API serves a similar purpose to the `RNif` and `RGenericAgent` APIs in v7.0, which were removed in v9.1.

`RConnection` is discussed in more detail in section 6.3.1. In practice, an application should consider an `RConnection` to represent a handle to a network connection. Note, that we said a handle to the connection, not the connection itself. Multiple applications may have handles to the same network connection.

6.2.2 The Default or ‘implicit’ Connection

Each Symbian OS device has a default network connection configured. This can be used by legacy applications (created before the `RConnection` API was introduced in Symbian OS v7.0s), or applications using the `stdlib` or PIPS sockets implementations (which are layered on top of `RSocket`), which do not use an `RConnection` – in this case the default network connection is started automatically if IP data is sent without first having connected an IAP explicitly. This is known as the *implicit* connection. The IAP used for the implicit connection is configured in `CommsDat` as part of the Connection Preferences table (again, discussed in section 6.3.2). UIQ allows the user to configure which IAP is used for the default connection as part of the internet account settings on the device, and it is presented to the user as the ‘Preferred’ internet account. The user also has an option to decide whether to be prompted to select an IAP when a connection is requested by an application that does not specify a connection itself, or whether the default IAP should always be used. For most applications on S60 the user is always prompted to select the IAP at connection time. This allows the user to have control of which connections are made for each application. However, it may not always be appropriate for the user to make this decision since not all applications will work on all networks (for example, MMS will only work on the network operator’s MMS network, which is why S60 allow the user to specify a single IAP for this in the messaging settings). Therefore if your application needs to connect over a specific network it is suggested that you provide a configuration option to the user to allow them to specify a network that should always be used. This gives your application full control over its choice of network connection.

6.3 Network Bearer Technologies in Symbian OS

Symbian OS supports the following IP bearer technologies, although configuration options may not be available for all of these on a given UI platform:

- Packet-switched data connections over GPRS/W-CDMA – provides operator services such as Internet access, MMS, music and video streaming.

- Circuit-switched data calls – provides Internet access via dial-up ISPs, access to private networks.
- Bluetooth PAN – provides a virtual Ethernet network over Bluetooth. Can be used for multi-player games over ad-hoc local networks. From v9.1 onwards Symbian OS supports the PAN-User (PANU) and Gateway Node (GN) roles. The GN role allows the phone to act as an Ethernet switch to connect other Bluetooth PAN devices together to create an ad-hoc network. The PANU role allows the device to connect to other Bluetooth PAN devices, and also to network access points to use external IP networks.
- Ethernet-based technologies such as wireless LAN. However not all devices provide WLAN support so check with the device manufacturers for a list of their WLAN-enabled devices.

Note: many existing IP-based applications on mobile phones are designed to use GPRS. Code examples in this chapter will also use GPRS, with an indication given where other technologies function or perform differently.

Note that Bluetooth PAN profile support is not available in S60 3rd edition devices.

Symbian OS makes connecting to different types of network as transparent as possible. However, the application developer should remember that different technologies have different performance characteristics, and a application that works well over WLAN may perform poorly over GPRS or a slower connection. This chapter will address how applications can make themselves robust to different network characteristics that may not be encountered on a device with a fixed network connection. Particular problems encountered on mobile devices include:

- Out of coverage – a network may be temporarily unavailable due to geographical conditions. Sometimes coverage can fluctuate over a matter of minutes, so this condition can range from being very transient to being more permanent (e.g., a number of hours, or possibly days).
- Limited resources – a network may be unavailable because the maximum number of connections already exists. Some examples of this are network operators supporting a maximum number of active PDP contexts per device, or if a WLAN connection is already active on the device, and the app wishes to connect to a different WLAN.

- Network busy – the network has no resources available to support more users.
- Service unavailable – whilst the network may be available and is supported by the device, a service may be unavailable at certain times, for example, when roaming, or due to lack of credit or no subscription.
- The time taken to make the network connection, particularly if user interaction is required.

The next section explains how to use the networking APIs to configure, connect and use IP networks. This can be briefly summarized as follows:

1. **Configure** – ensure the connection parameters are configured.
2. **Connect** – ensure the correct connection type and parameters are used.
3. **Use** – send and receive data via sockets.
4. **Disconnect** – indicate that you have finished with the connection.

6.3.1 Connecting to an IP Network

Applications wishing to create and use a IP network connection must have the `NetworkServices` capability. `RConnection` is used to connect to a specific IP network via an IAP as follows.

First, create a server connection to the sockets server:

```
RSocketServ iSockServ;          // this should be a member variable
                                // of a class
...
 TInt err = iSockServ.Connect();
```

As with all client/server connections, the error must be checked and errors handled.

Next, use this to open the `RConnection` object:

```
RConnection iConn;              // this should be a member variable
                                // of a class
...
err = iConn.Open(iSockServ);
```

Programmatically, there are at least three different ways to start a network connection, depending on your application's needs. Each of these may cause different user visible behavior, such as triggering a prompt asking which connection to use.

Without using RConnection

Applications can use sockets directly without having first explicitly started the network connection. When IP traffic is sent, the network connection will automatically be brought up. The user may be prompted to select which IAP is used in this case – this happens for most built-in applications on S60 devices but the user can override this behavior on UIQ and use the default (a.k.a. implicit or ‘preferred’) IAP.

The disadvantage of not using `RConnection` is that you have no way of tracking the status of the connection, so it is harder to determine if or why the connection has failed – errors returned from socket operations will provide an indication, but only some time later. Also, creation of a network connection may take a potentially large number of seconds, or on occasions a number of minutes, to succeed or fail (particularly if user input is required) so the application may try to send data before the connection has been fully established. This can cause unreliable data to be lost while the network connection is established – if this is important the `KSoUdpSynchronousSend` ioctl can be used, as described later in the chapter.

Using RConnection::Start() with no parameters

```
iConn.Start(iStatus);
```

In terms of selecting which IAP to start this is similar to not using an `RConnection`; however it gives the application more control and visibility of the connection as it is established – the request completes when the connection succeeds or fails and the application can also use the same `RConnection` to track the progress of the connection as described later. Because no connection parameters are given, it has the same user visible results as not using `RConnection` at all, i.e. the user may be prompted to select which IAP to use, or the default IAP will be used. If the connection fails, the error is returned asynchronously via the `TRequestStatus`. Again because the creation of the network connection may take a number of seconds the use of the asynchronous version of this API is recommended.

Using RConnection::Start() with an explicit IAP

To connect to a particular network, the application must provide an IAP ID in its call to `RConnection::Start()`. To do this, the application must first determine which IAP to use – this may already be configured by the user in the application’s own UI, or the application may use the `Comms-Dat` APIs to search for the most appropriate IAP of all those configured based on some application specific criteria. Either way, the application

will need to be able to retrieve IAP information from CommsDat. This is described in section 6.3.2.

6.3.2 Choosing the Connection Parameters

It is important to make sure that the correct connection parameters are used or the application may fail to work, the user may be over charged or private data may be sent over a public network. On a Symbian OS based device there are several networks available, potentially simultaneously. It should not be assumed that any network connection is suitable, especially as some network connections, such as the one to the operator's MMS network, are only suitable for a limited number of services – in this case, just the MMS service.

Connection configuration parameters are stored in CommsDat.

Previous versions of Symbian OS used CommDB; and the CCommsDatabase APIs are provided in order to support older applications and ease porting, but will be deprecated in future versions of Symbian OS. New applications should use the CommsDat APIs directly so only these are discussed.

Most devices are preconfigured with a set of parameters for connecting to the network operator's services, and also possibly for using BT PAN and WLAN for ad-hoc connections. Each application may need to use a different set of criteria for determining which IAP to use (e.g. user selected, based on bearer type etc.) but is likely to need to extract the relevant information about the currently configured IAPs from CommsDat to display to the user, and to use when starting the connection.

Figure 6.4 shows a simplified view of how the settings are stored in CommsDat. The full set of parameters is described in more detail in the Symbian OS Library, and is programmatically described in `CommsDat-TypesV1_1.h`.

The main table of interest is the IAP table – this contains the name of the IAP, as you would expect to see it in the device's UI, and links to the appropriate service and bearer tables for that IAP.

The per-technology service tables contain settings that vary between different services of a particular type – for example, the APN for a GPRS connection, the phone number for a CSD connection or an SSID for a WLAN connection, as well as the usernames and passwords for the service, and the IP addressing configuration (static, dynamically negotiated during connection, via DHCP, etc).

The per-technology bearer tables contain information that is fixed for accessing a particular type of service and often relates to the use of particular hardware – for example, the AT commands to send to a modem;

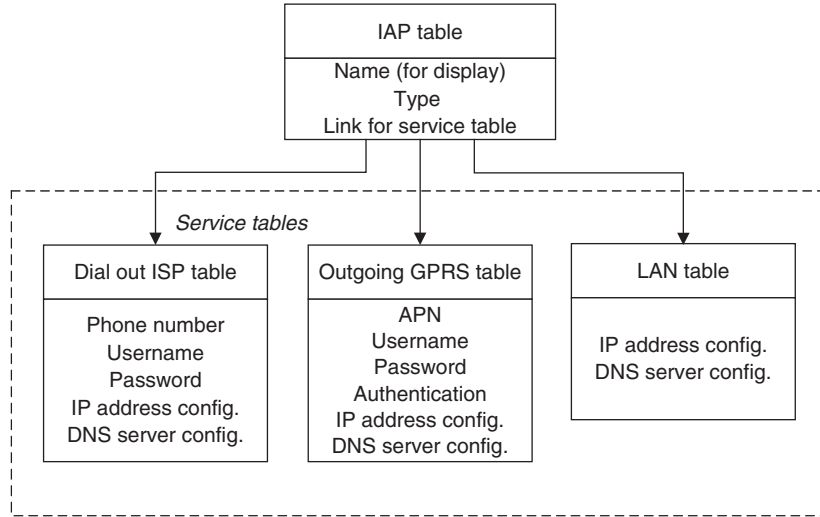


Figure 6.4 Connection parameters and IAPs stored in CommsDat

the NIF and agent name to use with a particular bearer; information on the hardware resources a connection should use, such as serial ports; and the values for the various modes of the idle timer, the purpose of which is explained further in section 6.3.4.

So for the application, the first thing to do is to create the CommsDat database session:

```
using namespace CommsDat;
CMDBSession* dbs = CMDBSession::NewLC(KCDVersion1_1);
```

Entries in CommsDat are accessed using record sets – `CMDBRecordSet` – of a given type. For the IAP table, this type is the `CCDIAPRecord`, identified by the `KCDTIdIAPRecord` record type ID. So to create and load a record set for the IAP table:

```
CMDBRecordSet<CCDIAPRecord> * iapTable = new (ELeave)
CMDBRecordSet<CCDIAPRecord> (KCDTIdIAPRecord);
CleanupStack::PushL(iapTable);
// Load the table into memory
TRAPD(r, iapTable->LoadL(*dbs));
```

Errors here could mean that the IAP table doesn't exist, i.e. that there are no IAPs configured. Applications should ensure that the user has configured an appropriate IAP.

Once the table is loaded, the `iRecords` member of the record set, an `RPointerArray`, can be used to iterate through the records in the table. Each member of the array is stored as a `CMDBRecordBase*` so

the returned object needs to be cast to its correct type, in this case a `CCDIAPRecord`. The process by which the application determines which IAP to use is up to the application – the example below extracts the name and type of the IAP for displaying to a user, and then stores the chosen IAP ID for use later. The full set of information available about an IAP is found in `CommsDatTypesV1_1.h`.

No platform security capabilities are required to read most entries from `CommsDat`, apart from fields considered sensitive, such as passwords, which typically require `ReadUserData`; or device sensitive data, which requires `ReadDeviceData`.

```
TInt numIaps = iapTable->iRecords.Count();
TInt chosenIAP = KErrNotFound;
for(TInt i = 0; i<numIaps; i++)
{
    CCDIAPRecord* iap = static_cast<CCDIAPRecord*>(iapTable->iRecords[i]);
    chosenIAP = iap->RecordId();

    if(iap->iBearerType.GetL().Compare(KCDTypeNameLANBearer==0)
    {
        TBuf<KMaxMedTextLength> buf(iap->iRecordName.GetL());
        // Add to buffer for display of ID and Name to user

        ...

        // This is the one we want
        break;
    }
}
```

This example iterates through all records looking for one's that use the `LANBearer` table for demonstration purposes, but there is a more efficient way to do this – using `FindL()`. To do this, a 'priming record' is created which contains the information against which to match.

```
// To start the search, create a new record set, and add a record
// containing the information to match against.
CMDBRecordSet<CCDIAPRecord*> iapRecordSet = new (ELeave)
    CMDBRecordSet<CCDIAPRecord>(KCDTIdIAPRecord);
CleanupStack::PushL(iapRecordSet);
// Match against LAN Bearers.
LIT(KBearerType, "LANBearer");
TPtrC bearerType(KBearerType);
// Create the record to fill with the data to match
CCDIAPRecord* primingRecord = static_cast<CCDIAPRecord*>
    (CCDRecordBase::RecordFactoryL(KCDTIdIAPRecord));
primingRecord->iBearerType = bearerType;
// Add it to the new record set
iapRecordSet->iRecords.AppendL(primingRecord);
// Ownership is passed to the recordset
primingRecord = NULL;
```

```
//Search
if(iapRecordSet->FindL(*iDb))
{
    //The iapRecordSet->iRecords.Count() will now reflect the number of
    // records found
    TInt iapRecordsFound = iapRecordSet->iRecords.Count();
    // The records can now be accessed as before...
}
```

When searching through records in CommsDat, it is more efficient to use Record IDs than names.

The type of the IAP is stored as a field in the record set, but to access the user settings the information in the service table is required.

All objects must be cleaned up once finished with:

```
CleanupStack::PopAndDestroy(2); // iapTable, dbs
```

Information about *active* IAPs (i.e. those which are currently connected) can also be retrieved via the RConnection API without needing to use the CommsDat APIs. This is discussed later in section 6.5.3.

Connection preferences

Once the correct IAP has been selected, a connection preference object, a TCommDbConnPref, must be created to store the IAP ID and other information (defined in CommDbConnPref.h):

```
TCommDbConnPref iConnPref // class member;
if(chosenIAP != KErrNotFound)
{
    iConnPref.SetIapId(chosenIAP);
}
```

In some cases, the IAP ID is all that needs to be provided, however, if you are explicitly providing an IAP it also pays to ensure that any prompting of the user is disabled.

```
iConnPref.SetDialogPreference(ECommDbDialogPrefDoNotPrompt);
```

The different possibilities for prompting are defined in TCommDbDialogPref:

```
/** Preference set to prompt user. */
ECommDbDialogPrefPrompt,
/** Preference set to warn user. */
ECommDbDialogPrefWarn,
/** Preference set not to prompt user. */
ECommDbDialogPrefDoNotPrompt,
/** Preference set to prompt user when in wrong mode. */
ECommDbDialogPrefPromptIfWrongMode
```

To limit the set of available IAPs selectable by the user to a particular set of technologies, you can also configure the bearer set in the connection preference:

```
connPref.SetBearerSet(ECommDbBearerWcdma | ECommDbBearerWLAN |
ECommDbBearerPAN )
```

Where the available subset is:

```
ECommDbBearerCSD
ECommDbBearerWcdma or ECommDbBearerGPRS - these are equivalent
ECommDbBearerPSD
ECommDbBearerVirtual
ECommDbBearerPAN
ECommDbBearerWLAN
```

If a particular technology is unavailable, then the system will automatically remove it from the available options.

6.3.3 Starting the Connection

Once the connection preferences have been configured, they can be used to start the network connection. We have already discussed starting the connection without using connection parameters like so:

```
iConn.Start(iStatus);
```

Using the connection preferences gives the application control over which IAP is selected and over the user interaction (prompting etc.) like this:

```
iConn.Start(iConnPref, iStatus);
```

As stated earlier, if the connection fails the error is returned asynchronously via the `TRequestStatus`. If the connection is started successfully, this will complete with `KErrNone` once the connection has reached the `KLinkLayerOpen` stage. More detailed information about the state of the connection can be gathered using the progress-reporting APIs discussed in section 6.5.1.

Put together, the code to start a known IAP explicitly looks like this:

```
// these are typically member variables of a class
// this is important, as the Start() call is asynchronous so
// iConnPref must not be a local variable
RSocketServ iSockServ;
RConnection iConn;
TCommDbConnPref iConnPref;

CSomeClass::StartConnectionL(TInt aChosenIap)
{
```



```

User::LeaveIfError(iSockServ.Connect());
User::LeaveIfError(iConn.Open(iSockServ));

iConnPref.SetIapId(aChosenIap);
iConnPref.SetDialogPreference(ECommDbDialogPrefDoNotPrompt);

iConn.Start(iConnPref, iStatus);
SetActive();
}

```

Setting multiple IAPs

It is possible for an application to provide a set of connection preferences to be tried in succession until one succeeds. This is done using the `TCommDbMultiConnPref` class. The following code will iterate through the IAPs with IDs 1, 2 and 3:

```

TCommDbConnPref connPref1;
connPref.SetIapId(1);

TCommDbConnPref connPref2;
connPref.SetIapId(2);

TCommDbConnPref connPref3;
connPref.SetIapId(3);

TCommDbMultiConnPref iMultiConnPref;
iMultiConnPref.SetPreference(1, connPref1);
iMultiConnPref.SetPreference(2, connPref2);
iMultiConnPref.SetPreference(3, connPref3);
iMultiConnPref.SetConnectionAttempts(3);

```

Supporting different bearers

One of the important things to remember is that if your application does not know or care how it connects to a network (i.e., it does not care which IAP is used), then it must be able to handle the different characteristics that different bearers display. In particular, any timers relating to sending/receiving data must be very carefully chosen, although our advice is to dispense with them altogether and rely on network layer timeouts where possible. Obviously for unreliable protocols like UDP this is not possible; however for TCP there should be no need for an application to limit the duration of a socket operation using its own timer. Any dependencies on data rates or network latencies must be carefully handled. In summary, remember not to depend upon certain behavior patterns just because they are exhibited by one type of network, and avoid building behavior that relies on certain network performance characteristics into the application wherever possible. This is particularly important with GPRS networks, where latencies can be in excess of 500 ms.

You should also be aware that there is a timer with a number of modes that measures how long a connection has been inactive, and can terminate a connection after a period of inactivity – section 6.3.4 contains more

details on this. You should be aware that the connection may be terminated for a number of reasons, one of which is the expiry of the idle timer.

Using an existing connection

In most cases, if you wish to have your application use an existing connection, it should call `RConnection::Start()` with the appropriate IAP information, just as it would if the connection were not already started. This increments the usage counter on the connection to ensure that it is aware of the new client.

The `RConnection::Attach()` method can be used to join an existing connection – however, it is far more simple just to use `Start()` as detailed above, as this works whether or not the connection is already started.

6.3.4 Stopping the Network Connection

There are a number of different ways for a network connection to be terminated. Of these, forced disconnection is the least desirable due to its potential for disrupting the operation of other applications on the device. In most cases, you should indicate that you do not need the connection any longer, then let the system arbitrate between applications and eventually terminate the connection.

Forced disconnection

`RConnection::Stop()` allows an application to terminate a network connection immediately. This should be rarely, or ideally never, be used as there may be other applications using the same connection, who will have their connection terminated. `Stop()` also requires the `NetworkControl` capability, which is part of the manufacturer capability set, due to its potential to disrupt the operation of other applications on the device.

```
err = iConn.Stop();
```

This version of `Stop()` causes sockets using the connection to be errored with `KErrCancel`. There is another version – used by the connection manager applications on the UI platforms – that causes `KErrRConnectionTerminated` to be returned instead. In this case the user has explicitly terminated that connection and therefore if you receive this error code from a connection you are using, you should not attempt to restart it.

Note that `Stop()` can return an error, which may happen, for example, if the connection has not yet been started (`KErrNotReady`) or if the

connection is active but cannot be stopped in its current state (`KErrorPermissionDenied`).

Automatically by the system due to lack of use

This is the preferred way to let connections shut down, as it is based on the usage of the connections by all applications. It is controlled by a series of timeouts set for the connection in `CommsDat`.

There is one timer for each connection that measures how long a connection has been idle, and depending on the current usage mode of the connection can be in one of three states. The timeout values for each of the three states are configured in `CommsDat` and one or more of them may be visible to the user – how they are exposed depends on the UI platform. The application does not need to be concerned with their values as long as it is able to handle the connection being terminated (and since you need to cope with the case where the user terminates the connection manually in your application, this case is not very different). The three different modes for the timer are based upon:

1. *When no user data has been sent for a given time.* On some UI platforms this time interval is configurable by the user, potentially only for some bearers, and the value usually differs between bearers. For example, a GPRS connection could be ‘always on’ without incurring any extra cost to the user, so would typically not have this timeout set; but a circuit-switched data call may be charged per-minute so could be terminated after five minutes of inactivity. On both S60 and UIQ, the timer is only exposed in the UI for circuit switched calls – other technologies have an infinite setting for this timer mode. This timer mode is called `LastSocketActivityTimeout` in bearer tables in `CommsDat`.
2. *When there are no open sockets or host resolvers using the network connection.* This time interval is usually specified by the phone manufacturer, and not exposed in the UI. It is called `LastSocketClosedTimeout` in the bearer tables in `CommsDat`.
3. *When there are no open `RConnection` objects using the connection.* `RConnection::Close()` allows an application to indicate that it no longer needs to use a network connection. This should be used in preference to `Stop()`, for the reasons indicated previously. Each connection keeps a reference count of applications (actually `RConnections`) that are using that connection, and once this number reaches zero the timer switches to this mode. In the bearer tables in `CommsDat` it is called `LastSessionClosedTimeout`.

Note that calling `RConnection::Close()` also closes the `RConnection`’s handle to the socket server, and hence all the `RSocket/`

`RHostResolver` objects that have been associated with that `RConnection` should be closed first. Also note that this `RConnection` cannot be used again until you have called `Open()` on it.

Note that in the case of (1) and (2), the application will still have a valid `RConnection`, which can be used to receive progress notifications, and also restart the connection if necessary.

Applications affect the lifetime of a network connection by the type of activity they perform or the objects they keep open – sockets, host resolvers or connections. In addition, the behavior of a connection depends on the values set for the different idle timer modes in `CommsDat`. If you wish to keep a connection open, it may be necessary to implement some form of keep-alive mechanism, depending on the values of the various timer modes and the network usage profile of your application. However, you should carefully consider the impact on both power consumption of the device and the potential to increase costs for the user.

Bearer-initiated disconnections

A connection to a mobile network may be terminated for a variety of reasons which are not related to the applications using it. For example, a GPRS network can initiate a disconnection from the network side due to lack of resources, network errors, or idle timeouts in the network; or the device may go out of range of the network. These are also common errors for other wireless bearers such as Bluetooth or WLAN. Applications need to be designed to cope with this and handle the errors cleanly, e.g. by retrying the connection a finite number of times, or by prompting the user to retry later. The application must not repeatedly attempt to restart the connection ad infinitum because that will quickly drain the battery of the device.

Some errors are fatal whilst others are not, so applications should be able to determine when a retry mechanism has a chance of succeeding and when to just notify the user about the problem – typically errors about transient failures such as `KErrNotReady` should result in a limited number of retries, whereas permanent errors, such as `KErrAccessDenied` should result in an immediate user notification. This can be difficult in some circumstances, as the application isn't always aware of which bearer is being used (WCDMA, WLAN, etc.) and the errors may differ depending on the bearer. It is possible to retrieve the last error that occurred on a connection using `RConnection::LastProgressError()`.

6.4 Using the Network Connection

Once the application has a handle to an active network connection, it can send and receive IP data. The first thing it might want to do is to resolve

a domain name into an IP address. Before domain name resolution is discussed, internet addressing on Symbian OS needs some explanation.

6.4.1 Internet Addresses

IP addresses on Symbian OS are represented by the `TInetAddr` class. This is a derivation of the generic `TSockAddr` class already discussed in Chapter 3. A `TInetAddr` stores both the IP address and the TCP/UDP port number to be used for the socket. Symbian OS supports both IPv4 and IPv6 addresses, defined in `in_sock.h`. The address family can be determined using the `Family()` method of the underlying `TSockAddr` class. This will be `KAfInet` for an IPv4 address, and `KAfInet6` for an IPv6 address:

```
if( addr.Family() == KAfInet)
{
    // IPv4 addressing
}
else if (addr.Family() == KAfInet6)
{
    // IPv6 addressing
}
```

Addresses returned from the networking subsystem are usually in `KAfInet6` format so care must be taken by the application if it intends to access the data in an address (rather than just passing it from `RHostResolver::GetByName()` to `RSocket::Connect()`) to handle these using the correct class as described below. Currently, many addresses, although returned in IPv6 format, are actually IPv4 address in IPv4-mapped format inside the IPv6 address.

IPv4 addresses

IPv4 addresses are stored as a 32-bit integer. However, they can be handled by the application in a number of different formats and conversion methods exist to translate between them:

- In 'dotted quad' notation inside a descriptor, for example "192.168.1.1". In this case the `Input()` method is used to convert from the descriptor to the `TInetAddr`:

```
_LIT(KInetAddr, "192.168.1.1");
TInetAddr addr;
addr.Input(KInetAddr);
```

- As a 32-bit integer.
- As four separate 8-bit integers, each representing part of the address.

The `INET_ADDR` macro makes these last two formats easy to use, and the `SetAddress()` method can be used:

```
const TUint32 KInetAddr = INET_ADDR(192,168,1,1);  
addr.SetAddress(KInetAddr);
```

The address can also be created using the constructor if the port number is also known:

```
const TUint32 KInetAddr = INET_ADDR(192,168,1,1);  
const TInt KPortNum = 8080;  
TInetAddr addr(KInetAddr, KPortNum);
```

IPv6 addresses

IPv6 addresses are handled in much the same way as IPv4 addresses, but because they are 128 bits long, they are stored inside a `TSockAddr` as a `TIp6Addr` object. `TIp6Addr` can also store IPv4 addresses. Sockets require a `TSockAddr`-derived object so when using IP addresses directly, rather than obtaining them from the DNS, the `TInetAddr` must be created using the `TIp6Addr` in the same way as the IPv4 address is created using the IPv4 specific overloads.

All addresses are handled in the native byte order, so no byte-order conversion functions are necessary.

Useful constants

There are also several commonly used IP addresses predefined in Symbian OS that applications may find useful such as `KInetAddrAll` and `KInetAddrBroadcast` (both 255.255.255.255) and `KInetAddrLoop` (127.0.0.1). Others are defined in `in_sock.h`.

6.4.2 Domain Name Resolution

Domain name resolution is carried out using the `RHostResolver` API. This has already been discussed in Chapter 3 but is discussed specifically in the context of DNS here. DNS is used to resolve an internet domain name into an IP address via a DNS server, as defined in RFC1034 and shown in Figure 6.5. Applications do not need to know the specifics of how this works, but they should be aware of the configuration and behavior of the DNS implementation on Symbian OS.

Each network interface is configured with the IP address of one or more DNS servers, and in most cases this is provisioned dynamically during the connection process. Alternately, it can be configured statically as part of the appropriate record in a service table in `CommsDat`. To ensure that the DNS query is sent to the correct DNS server, an `RConnection` must be associated with the `RHostResolver` when it is opened. Legacy applications, or applications using `stdlib` or `PIPS` sockets, and hence

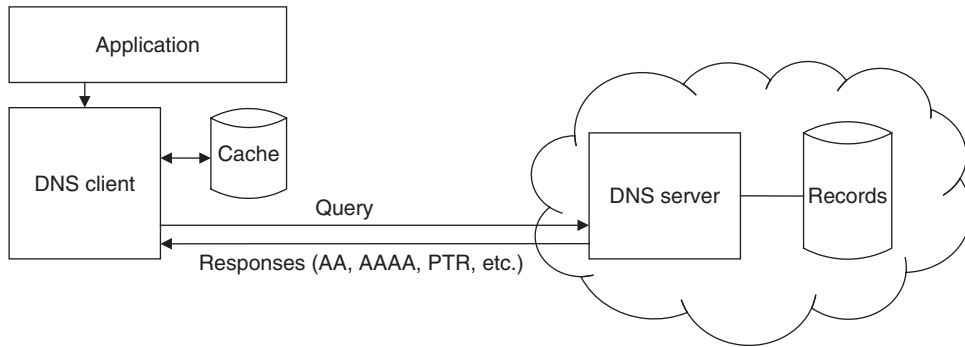


Figure 6.5 DNS architecture

not using `RConnection` will send the DNS queries over the implicit connection, which will create a network connection if there is not a suitable one active. Once a valid DNS response is received, it is cached until the lifetime of the response is reached, the device is shutdown, or the cache is cleared.

```

RHostResolver iHr; // typically class members of an active object
TNameEntry iEntry;

...
_LIT8(KWwwSymbianCom, "www.symbian.com");
User::LeaveIfError(iHr.Open(iSockServ, KAfInet, KProtocolInetUdp, iConn));
hr.GetByName(KWwwSymbianCom, iEntry, iStatus);
SetActive();
  
```

Responses are stored in the `TNameEntry`. On successful completion, the IP address will be stored in the `iAddr` member of the name entry:

```

TInetAddr addr = iEntry.iAddr;
  
```

If there are multiple addresses or aliases, they can be retrieved using `Next()`:

```

iHr.Next(iEntry, iStatus);
  
```

until the error `KErrEof` is returned to indicate that there are no more valid addresses. Note that it is also possible to receive errors here if, for example, the network connection is terminated while a query is ongoing, or the network connection has not started. Specific DNS error codes include:

- `-5120`, indicating the DNS server could not be contacted. This is often because the connection has not started, but can also be because the configured DNS server address is wrong.

- -5121, indicating that the host is not found in the DNS records. This normally indicates that the user has mistyped the hostname.

TNameEntry contains more than just an IP address – it also contains flags in the iFlags variable indicating the origin of the response. These can be one or more of the following:

EDnsAlias	Answer is an alias
EDnsAuthoritative	Answer is authoritative
EDnsHostsFile	Answer is from host's file
EDnsServer	Answer is from a DNS server
EDnsHostName	Answer is host name for this host
EDnsCache	Answer is from the resolver cache

The first response to GetByName() is the matching IP address (A(IPv4) or AAAA(IPv6) records, depending on what the DNS server returned). Subsequent calls to Next() will return any further aliases (CNAMEs) or IP addresses (A or AAAA records). Aliases will have the IP address part of the TNameEntry set to 0 and the EDnsAlias flag set.

DNS also enables applications to resolve an IP address into a domain name, using the GetByAddress() method, assuming that the reverse lookup (or more precisely the PTR record) is set up correctly – which for many Internet hosts is often not the case:

```
TInetAddr iAddr; // usually members of an active object
TNameEntry iEntry;

...
const TUInt32 KInetAddr = INET_ADDR(192,168,1,1);
iAddr.SetAddress(KInetAddr);
iHr.GetByAddress(iSockServ, iAddr, iEntry, iStatus);
SetActive();
```

On successful completion, the name will be stored as a descriptor in the iName of the TNameEntry:

```
THostName name = entry.iName;
```

As for GetByName(), the first response to GetByAddress() will be the first matching domain name (PTR record). Subsequent calls to Next() will return any further matching domain names, again until KErrEof is returned.

Note that for all DNS-based `RHostResolver` queries, `Next()` can be called synchronously because all the information is retrieved during the initial query and stored in a local buffer.

6.4.3 Sending and Receiving Data

At this point your application has an IP address of a peer to which to send data. In most cases, the application will be using the TCP or UDP protocols. It is also possible to transmit raw IP and ICMP but these aren't discussed here, especially as the `NetworkControl` capability is required to access IP and ICMP sockets. Symbian OS's socket APIs are based on BSD sockets and should be a familiar paradigm to developers used to socket implementations on other operating systems. Chapter 3 discusses Symbian OS sockets in more detail, so this chapter only discusses how they are used for IP data. Figure 6.7 shows the flow of data through the IP stack.

Whichever IP-based protocol is used, each `RSocket` should be bound to the `RConnection` over which it is intended to be used.

TCP

Figure 6.6 is a simplified version of the TCP state machine, showing the states in TCP that are triggered using methods on the `RSocket` API.

Chapter 3 has example code for creating and connecting a TCP socket for both incoming and outgoing connections. Once the TCP socket is connected, data transfer is possible.

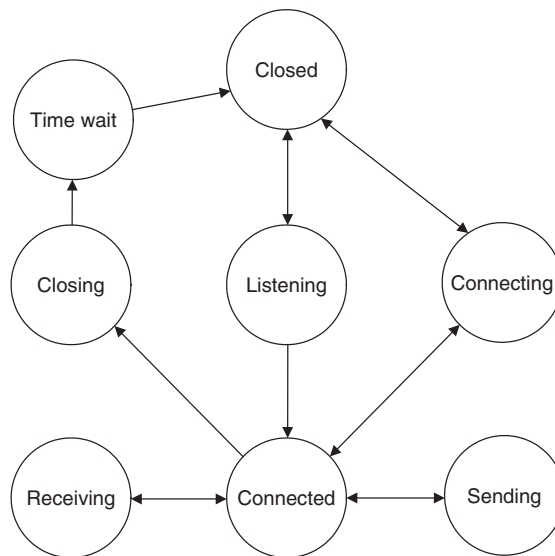


Figure 6.6 TCP state machine

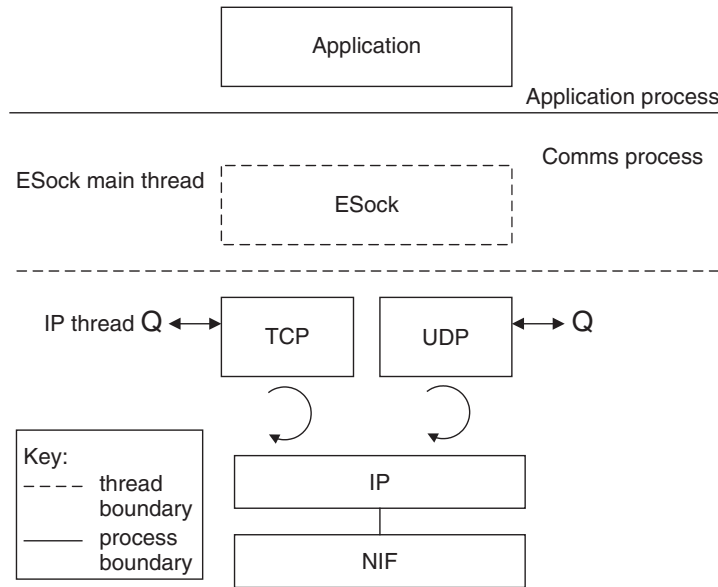


Figure 6.7 Networking subsystem data handling

TCP data transfer

For TCP, a call to `Write()` or `Send()` will complete once the data has been queued by the TCP/IP stack. The application can be notified of when the data is actually sent from the device using the `KIOctlTcpNotifyDataSent` socket ioctl (again, Chapter 3 discusses ioctls in more detail):

```
sock.ioctl(KSolInetTcp, KIOctlTcpNotifyDataSent, stat);
```

However, as we discussed in Chapter 3, knowledge of when data has left a device is less useful than it appears. There is no guarantee that data is going to make it across the network without one or more retransmissions, and even then there is no guarantee that even though the remote device has received the data that it has been passed to the remote application. As a result of this, we recommend that any protocol that requires knowledge of whether a particular operation succeeded or failed on a remote host include some form of application level response code system – as is used in HTTP, for example, with response codes being sent for every request.

To receive data, use either `Recv()`, `Read()`, `RecvOneOrMore()` or `RecvOneOrMore()`. These complete at different times with different amounts of data, as described in Chapter 3, so the application will need to determine which to use depending on its needs.

Terminating the TCP connection

To terminate the TCP connection, either `Shutdown()` or `Close()` needs to be used. `Shutdown()` is asynchronous, and performs a clean shutdown of the socket connection when called with `ENormal`, so this is the best method to avoid blocking the thread while the TCP connection is terminated. However, there are a number of other ways to terminate the connection by passing different parameters to `Shutdown()` as described in Chapter 3. `Close()` will shutdown the connection and also simultaneously close the socket handle, and is a synchronous call. By default, if `Close()` is called without first calling `Shutdown()` it behaves like the `ENormal` variation of `Shutdown()`, unless additional data is received from the remote device after `Close()` has been called in which case it will behave like an `EImmediate` Shutdown. This behavior can be modified by using the `KSoTcpLinger` socket option with the `linger` option turned on. This can ensure that `Close()` returns immediately, but with the TCP disconnection continuing in the background.

```
TSoTcpLingerOpt lingerOpt;
lingerOpt.iOnOff = 0; // Off
sock.SetOpt(KSolInetTcp, KSoTcpLinger, lingerOpt);
```

Setting the `linger` option on causes `Close()` to block until the `linger` timeout specified has expired. After that time, it behaves as the default, and the disconnection will continue in the background.

```
lingerOpt.iOnOff = 1; // On
lingerOpt.iLinger = 2; // 2 seconds
sock.SetOpt(KSolInetTcp, KSoTcpLinger, lingerOpt);
```

Reconnecting to the same port on the same remote device after shutdown may fail while the TCP connection is in the `WAIT_STATE`. It is possible to use the `KSoTcpAsync2MslWait` socket option to wait for a connection to become available again.

Controlling the behavior of TCP operations

There are several other socket options and `Ioctl`s supported for TCP, the most commonly used or useful of which are outlined below:

Enable TCP keep alive:

```
sock.Ioctl(KSolInetTcp, KSoTcpKeepAlive, 1, iStatus);
```

The following options, are available through `RSocket::GetOpt()`:

- `KSoTcpSendBytesPending` – to discover the number of bytes in the TCP send queue.

- `KSoTcpReadBytesPending` – to discover the number of bytes in the TCP receive queue.

UDP

UDP is a connectionless datagram-based protocol and so doesn't involve a connection establishment to the peer device. The application just needs to specify the local and remote port numbers and IP address to send the data to. There is no state machine as such, certainly not of the complexity of the TCP state machine.

To open a UDP socket, the following parameters must be specified when the socket is opened:

- the Internet address family, `KAfInet`
- the socket type, `KSockDatagram`
- the UDP protocol family, `KProtocolInetUdp`

```
RSocket sock;  
err = sock.Open(iSockServ, KAfInet, KSockDatagram, KProtocolInetUdp,  
               iConn);
```

If a known port is to be used for receiving UDP packets, a local port needs to be bound to the socket. This can also be done dynamically if a fixed port isn't required (and the application can communicate this port number to the remote device via a higher layer protocol). A dynamically set port number can be retrieved using the `LocalAddr()` method. This is the same method as for TCP and errors should be handled in the same way in case the port is already being used by other applications.

```
addr.SetPort(8080);  
err = iSock.Bind(addr);
```

As with TCP, if the `RConnection` is not specified when the socket is opened, the implicit connection will be used. However, because UDP is a connectionless protocol there is no IP traffic sent during connection setup so the connection will not be automatically started, unless it has been triggered by another means, for example, a DNS lookup. If the application is expecting to receive UDP packets before sending any, it will need to explicitly start the network connection to ensure it is active. This is important, as no data will be received if the network connection isn't active! This can be done using the `RConnection` API as described earlier in the chapter, or alternatively the implicit connection will be started when either `Connect()` is called on a UDP socket or if data is written to the socket. Note that `Connect()` doesn't actually send any IP traffic.

```
const TUInt32 KInetAddr = INET_ADDR(192,168,1,1);
iAddr.SetAddress(KInetAddr);
iAddr.SetPort(8080);
iSock.Connect(iAddr, iStatus);
SetActive();
```

Applications that will only work on specific networks must supply an IAP ID as a connection preference to the `RConnection` API to make sure the correct IAP is started.

One advantage of using `Connect()` is that it allows the application to specify a single remote IP address up front so the `Send()` method can be used to send data instead of specifying the address in each send as part of the `SendTo()` call. But this may not be appropriate if there are multiple remote devices involved.

```
TSockXfrLength iLen = 0; // class member variable
...
TInt flags = 0;
sock.SendTo(KHelloWorld, addr, flags, iStatus, iLen);
```

The flags parameter is unused for UDP sockets, and for datagram sockets in general the `TSockXfrLength` parameter will contain the number of bytes actually sent once the send completes successfully. Applications using active objects must make sure that the `TSockXfrLength` has the same lifetime as the duration of the active object (i.e., it should be a class member, rather than a local variable).

Because UDP is an unreliable protocol, datagrams may be discarded before leaving the device, for example, if the internal queue is filled. However, it is possible for an application to enforce that a send operation blocks rather than allows the queue to fill if it would result in packets being discarded. An example of this is where the implicit network connection is started as the result of the application sending data – it could take a number of seconds for the network connection to activate, during which time datagrams may be discarded if the queue fills (the write will complete as soon as the data is queued, so the application may continue to write datagrams causing the queue to fill). To enforce that a send will block if the send queue is full so that datagrams aren't discarded, the application can set the `KSoUdpSynchronousSend` socket option:

```
sock.SetOpt(KSolInetUdp, KSoUdpSynchronousSend, iStatus);
```

Note, however, that apart from preventing datagrams being lost during connection startup, this option is of limited general use, as preventing the local device dropping datagrams still does not guarantee that they will not be discarded by the network or the remote device.

Receiving UDP datagrams is just as easy as sending. If `Connect()` has been called on the socket, the `Recv()` variation can be used, else `RecvFrom()` is needed so that the remote address is known. In this case, `Connect()` also acts to filter the received datagrams to ensure they are from the correct address.

```
sock.Recv(iBuf, flags, iStatus);
```

Again the `flags` parameter is not used. `Recv()` will complete once a datagram has been received. However, the receiving buffer must be large enough to hold the entire datagram else the remaining data will be lost. The version of `Recv()` using a `TSockXfrLength` is not valid for datagrams.

For an unconnected socket, applications need to use `RecvFrom()` to be able to extract the IP address of the sending device:

```
sock.RecvFrom(iBuf, iAddr, flags, iStatus);
```

On successful completion, the `iAddr` parameter will contain the IP address and port number of the sending device.

Note that it isn't possible (or sensible) to use `RecvOneOrMore()` with UDP sockets, or datagram sockets in general, because the receive only completes once an entire datagram has been received.

Cancelling asynchronous operations

For both TCP and UDP, all asynchronous operations can be cancelled using the relevant `Cancel...`() method. You should be aware that the usual rules for cancelling an asynchronous service apply though – namely that calling `Cancel()` is a declaration that you are not interested in the result of the operation, **not** a way of preventing the operation taking place. For example, your data may already have been placed in an outgoing queue by the time you call `Cancel()`, in which case all you're doing is preventing the notification of the completion of operation from occurring.

There is also a `CancelAll()` method which will cancel all outstanding asynchronous requests. This is particularly useful when destroying an object that can perform many different asynchronous operations on a socket without having to call all the individual `Cancel...`() methods in the destructor.

Getting the local IP address

One of the (potentially many) IP addresses of a local interface can be retrieved from a socket using the `LocalName()` method:

```
sock.LocalName(addr);
```

On a device that has many network connections active, this will return one of the IP addresses belonging to the connection associated with the `RConnection` that was used to open the socket (or of the implicit connection if no `RConnection` was used).

6.4.4 How to use Secure Sockets in Symbian OS

Secure sockets, or sockets supporting the secure sockets layer (SSL) and Transaction Layer Security (TLS), are supported on Symbian OS through `CSecureSocket` API. Secure sockets are familiar to most developers as the basis for `https://` connections. They provide a way for the client to authenticate the server to which they are connecting, and provide an encrypted channel to prevent eavesdropping and alteration of the data transferred. In some cases, secure sockets can also be used to prove the identity of a client to a server – however this usage is far less common due to the low number of client devices with an appropriate certificate that could be used to identify them.

The `CSecureSocket` API is used to secure an already open and connected socket. `CSecureSocket` is instantiated with a reference to an already connected `RSocket` and a relevant protocol name. The `CSecureSocket` finds, loads and creates a secure socket of the correct implementation.

The main part of the `CSecureSocket` API looks like this:

```
class CSecureSocket : public CBase
{
public:
    IMPORT_C static CSecureSocket* NewL(RSocket& aSocket, const TDesC&
        aProtocol);
    IMPORT_C void Close();
    IMPORT_C void Send(const TDesC8& aDesc, TRequestStatus& aStatus,
        TSockXfrLength& aLen);
    IMPORT_C void Send(const TDesC8& aDesc, TRequestStatus& aStatus);
    IMPORT_C void CancelSend();
    IMPORT_C void Recv (TDes8& aDesc, TRequestStatus& aStatus);
    IMPORT_C void RecvOneOrMore( TDes8& aDesc, TRequestStatus& aStatus,
        TSockXfrLength& aLen );
    IMPORT_C void CancelRecv();
    IMPORT_C void CancelAll();
    IMPORT_C void CancelHandshake();
    IMPORT_C const CX509Certificate* ClientCert();
    IMPORT_C TInt SetClientCert(const CX509Certificate& aCert);
    IMPORT_C TClientCertMode ClientCertMode();
    IMPORT_C TInt SetClientCertMode(const TClientCertMode aClientCertMode);
    IMPORT_C const CX509Certificate* ServerCert();
    IMPORT_C TInt SetServerCert(const CX509Certificate& aCert);
    IMPORT_C TDialogMode DialogMode();
    IMPORT_C TInt SetDialogMode(const TDialogMode aDialogMode);
    IMPORT_C TInt AvailableCipherSuites(TDes8& aCiphers);
    IMPORT_C TInt SetAvailableCipherSuites(const TDesC8& aCiphers);
    IMPORT_C TInt CurrentCipherSuite(TDes8& aCipherSuite);
```

```

IMPORT_C void StartClientHandshake(TRequestStatus& aStatus);
IMPORT_C void StartServerHandshake(TRequestStatus& aStatus);
IMPORT_C TInt Protocol(TDes& aProtocol);
IMPORT_C TInt SetProtocol(const TDesC& aProtocol);
IMPORT_C void RenegotiateHandshake(TRequestStatus& aStatus);
IMPORT_C void FlushSessionCache();
IMPORT_C TInt GetOpt(TUint aOptionName, TUint aOptionLevel,
                    TDes8& aOption);
IMPORT_C TInt GetOpt(TUint aOptionName, TUint aOptionLevel,
                    TInt& aOption);
IMPORT_C TInt SetOpt(TUint aOptionName, TUint aOptionLevel,
                    const TDesC8& aOption=TPtrC8(NULL,0));
IMPORT_C TInt SetOpt(TUint aOptionName, TUint aOptionLevel,
                    TInt aOption);
};

```

It should be noted that the Symbian OS TLS implementation currently only supports client mode, i.e. using the device as a client connecting to a remote server. Any CSecureSocket methods that require the device to act as a server for remote clients will return KErrNotSupported. These interfaces include SetServerCert() and StartServerHandshake().

The handshake

The first thing we need to do is create, open and connect an RSocket. TLS can be used to secure any connection-oriented socket. CSecureSocket requires an already open and connected socket as input.

```

RSocketServ iSockServ; // member variables of a active object
RSocket iSocket;
TInetAddr iDstAddr;

...
_LIT(KTestAddress, "192.168.1.1");
_LIT(KWwwTlsPort, 443); // port on which we'd expect to find
                        // servers running TLS to offer HTTP
                        // services

User::LeaveIfError(iSockServ.Connect());
User::LeaveIfError(iSocket.Open(iSockServ, KAfInet, KSockStream,
                              KProtocolInetTcp, iConnection));

//Connect the socket
iDstAddr.SetPort(KWwwTlsPort);
iDstAddr.SetAddress(KTestAddress);
User::LeaveIfError(iSocket.Connect(iDstAddr, iStatus));

```

Next we can create the secure socket by calling CSecureSocket::NewL() and passing in the RSocket and secure protocol name. The supported protocol names are "SSL3.0" or "TLS1.0" and can be supplied in upper or lower case.

The function will return a pointer to the newly created `CSecureSocket`. The dialog mode and current cipher suites are set to the defaults, which are Attended mode and {0x00,0x00} respectively.

```
CSecureSocket* iSecureSocket; // class member

...
_LIT(KSSLProtocol, "tls1.0");
iSecureSocket = CSecureSocket::NewL(iSocket, KSSLProtocol);
```

We can optionally specify which cipher suites we want to use in the negotiation. We would need to call `AvailableCipherSuites()` to retrieve all the cipher suites supported by the device, then choose a subset of cipher suites to use for the session. If a preference is not specified, the order of the cipher suites will be the default based on the available cipher suites.

The list of cipher suites supplied to `SetAvailableCipherSuites()` must be in two-byte format, i.e., {0x00,0x25}. The order of the suites is important, they must be listed with the preferred suites first.

The following cipher suites (in priority order) are supported by the current implementation:

Priority	Cipher suite	Value
1	TLS_RSA_WITH_AES_256_CBC_SHA	{0x00,0x35}
2	TLS_RSA_WITH_AES_128_CBC_SHA	{0x00,0x2F}
3	TLS_RSA_WITH_3DES_EDE_CBC_SHA	{0x00,0x0A}
4	TLS_DHE_RSA_WITH_3DES_EDE_CBC_SHA	{0x00,0x16}
5	TLS_DHE_DSS_WITH_3DES_EDE_CBC_SHA	{0x00,0x13}
6	TLS_RSA_WITH_RC4_128_SHA	{0x00,0x05}
7	TLS_RSA_WITH_RC4_128_MD5	{0x00,0x04}
8	TLS_RSA_WITH_DES_CBC_SHA	{0x00,0x09}
9	TLS_DHE_DSS_WITH_DES_CBC_SHA	{0x00,0x12}
10	TLS_RSA_EXPORT_WITH_DES40_CBC_SHA	{0x00,0x08}
11	TLS_RSA_EXPORT_WITH_RC4_40_MD5	{0x00,0x03}
12	TLS_DHE_DSS_EXPORT_WITH_DES40_CBC_SHA	{0x00,0x11}
13	TLS_DHE_RSA_EXPORT_WITH_DES40_CBC_SHA	{0x00,0x14}
14	TLS_DH_anon_EXPORT_WITH_DES40_CBC_SHA	{0x00,0x19}

Note, the following cipher suites are currently not supported:

- Diffie–Hellman (static) cipher suites
- Anonymous Diffie–Hellman
- SSL v3.0 Fortezza cipher suites.

```
const TUInt KCipherSuiteSize = 2 ;
TBuf8<KCipherSuiteSize> cipherSuite;
cipherSuite.SetMax();

cipherSuite[0] = 0x00;
cipherSuite[1] = 0x03;

// Set cipher suites
User::LeaveIfError(iSecureSocket->SetAvailableCipherSuites(cipherSuite));
```

Next, we need to set the domain name using:

```
RBuf8 iHostName;      // descriptor containing the name of the host to
                      // which we connected (class member)

...
err = iSecureSocket.SetOpt(KSoSSLDomainName, KSolInetSSL, iHostName);
```

To allow us to perform verification of the server certificate during the handshake negotiation. We should pass in a descriptor containing the name or IP address (depending on which the user entered) that we connect used to the socket. Note that if the user asked us to connect to a server by name, which we first had to resolve using `RHostResolver`, that we should pass that name into the `SetOpt()` call rather than the IP address we got back from DNS. This is because most certificates are issued to servers by name rather than by address, and we are verifying that the name in the certificate matches the name of the host to which we tried to connect.

Then we can initiate a handshake with the remote server, using `StartClientHandshake()`. This is an asynchronous call and will only complete when the handshake completes and the secure connection is established, or there was a failure during connection establishment. If there is pending data to read on the `RSocket` when this method is called, it will complete immediately with the error code `KErrSSLSocketBusy`.

```
// Start client handshake
err = iSecureSocket->StartClientHandshake(iStatus);
```

Typically, the client needs to authenticate the server, by checking the certificate the server provided to see if it chains back to a certificate held

in the local device's certificate store, and trusted for use with TLS. In addition to this, the server can request that the client also authenticate by providing a certificate, although this is much less common. In this case `ClientCert()` can be used to retrieve the client certificate if the server has requested one. This API will return a pointer to the current client certificate or a null pointer if no suitable certificate is available. A client certificate can only be returned after the negotiation is complete.

```
// Get the client certificate
CX509Certificate* clientCert = iSecureSocket->ClientCert()
```

Likewise, the client can retrieve the server certificate using `ServerCert()`. This returns a pointer to the current server certificate, if available. A server certificate can only be returned to the client after the negotiation has reached a stage where the certificate has been received and verified.

```
// Get the server certificate
CX509Certificate* serverCert = iSecureSocket->ServerCert()
```

It is possible to retrieve the current cipher suite from the server using `CurrentCipherSuite()`. This method can only return the current cipher suite when the server has proposed one, which is after the handshake negotiation is complete. If this method is called before completion of the handshake, the cipher values will be null, or more explicitly `[0x00][0x00]`.

```
// Get the current cipher suite
err = iSecureSocket->CurrentCipherSuite(cipherBuf);
```

At any time after the initial handshake completes, a renegotiation can be initiated using `StartRenegotiation()`. `StartRenegotiation()` suspends data transmission and reception and restarts the handshake negotiation. It also creates a new TLS provider object to access security services. This is necessary as a new cryptographic token might be selected to create new key material. The session cache is flushed to ensure a new session is created and that entirely new key material is generated.

```
// Renegotiate the handshake
iSecureSocket->StartRenegotiation(iStatus);
```

The handshake can be cancelled using the synchronous method `CancelHandshake()`. The state of the connection cannot be guaranteed,

therefore it is recommended to call `Close()`. `Close()` will close the secure connection by cancelling all outstanding operations and closing the socket.

```
// Cancel the handshake
iSecureSocket->CancelHandshake();
iSecureSocket->Close();
```

Data transfer

Once the handshake phase is complete, the secure socket is ready for data transfer. Only one send operation can be outstanding at a time. The `Send()` will complete with `KErrorNotReady` if called when another send operation is still outstanding.

```
// Send some data
iSecureSocket->Send(iBuffer, iStatus);
```

It is also possible to cancel the socket send by using `CancelSend()` as shown below:

```
// Cancel the Send
iSecureSocket->CancelSend();
```

To receive data from the socket, `Recv()` or `RecvOneOrMore()` can be used. `Recv()` will complete when the descriptor has been filled, while `RecvOneOrMore()` will complete when at least one byte has been read, although typically it provides more data than this. These functions are asynchronous and only one `Recv()` or `RecvOneOrMore()` operation can be outstanding at any time. More details and advice on the use of these methods is given in Chapter 3: it's worth paying particular attention to the fact that `Recv()` blocks until the supplied descriptor is filled.

```
// receive some data using Recv()
iSecureSocket->Recv(iBuffer, iStatus);

// receive some data using RecvOneOrMore()
iSecureSocket->RecvOneOrMore(iBuffer, iStatus);
```

To cancel a receive, `CancelRecv()` will cancel any outstanding read operation:

```
// Cancel the Receive
iSecureSocket->CancelRecv();
```

Useful utility interfaces

There are some methods in the `CSecureSocket` API that aren't strictly part of the handshake or data transfer. They are documented here for completeness.

FlushSessionCache() This is used to support abbreviated handshakes. This method is used as an indication that the client does not intend to reuse an existing session. As such it allows the client to specify whether a new or existing session will be used in the handshake negotiation. Note, however, that there is no means of indicating the success or failure of this operation to the client.

```
// Flush the session cache
secureSocket->FlushSessionCache();
```

SetDialogMode() and DialogMode() These methods are used to set and get the untrusted certificate dialog mode. It determines if a dialog is displayed when an untrusted certificate is received. There are two modes of operation that are specified as part of the `TDialogMode` enumeration:

- `EDialogModeAttended`, all untrusted certificates result in a user dialog.
- `EDialogModeUnattended`, untrusted certificates are cancelled without user confirmation.

```
// Set the dialog mode to EDialogModeAttended
err = iSecureSocket->SetDialogMode(EDialogModeAttended);
// check error code!

// Get the dialog mode
TDialogMode dialogMode = iSecureSocket->DialogMode();
```

SetProtocol() and Protocol() These synchronous methods get and set the protocol in use. Presently, this can be either "SSL3.0" or "TLS1.0".

When `Protocol()` is called before the handshake has taken place, the protocol name returned is the one that will be proposed in the connection. However, once the handshake has taken place, and the secure connection has been established, the protocol name returned is the one that is actually being used.

For instance, the "TLS1.0" protocol may be proposed, but if the handshake takes place and the remote end doesn't support TLS1.0, then the protocol used would be SSL3.0.

The `SetProtocol()` method sets the protocol version that will be proposed in the connection. Setting “TLS1.0” means that a connection can still fall back to SSL3.0 if required. Setting “SSL3.0” means that TLS1.0 will not be available for use, even if the remote end only supports TLS1.0.

```
// Set the protocol to TLS1.0
_LIT(KSSLProtocol, "tls1.0");
err=iSecureSocket->SetProtocol(KSSLProtocol);
// check error code

// Get the protocol
const TUInt KMaxTlsProtocolNameSize = 8;
TBuf<KMaxTlsProtocolNameSize> protocol;
err=iSecureSocket->Protocol(protocol);
// check error code
```

6.5 Information Gathering and Connection Management

`RConnection` also allows an application to gather information about the state of their network connection, as well as information about other network connections in the system. Probably the most important of these is the state of the current connection which is published through progress notifications.

6.5.1 Progress Notifications

It is possible to track the progress of the current network connection using `ProgressNotification()`. During the connection and disconnection procedure, the connection will go through various stages each of which will produce a notification of progress. This is extremely useful, as it allows a user interface to give feedback to the user on the current status of the connection. Whilst connection setup may be a long-running operation, each individual step is considerably shorter. This means that the user can be provided with much better feedback that the connection establishment is making progress rather than stalled. Applications can drive these UI updates by subscribing to progress notifications and updating the UI as each new progress notification arrives.

Each bearer technology provides its own set of progress notifications for any bearer specific states, but there are also generic stages which will be common to all bearers. The generic stages, and the transitions between them are shown in Figure 6.8.

These generic stages are defined in `nifvar.h`:

- `KConnectionUninitialised` – the start and end stage. The connection has not yet been attempted or a previous connection has been terminated.

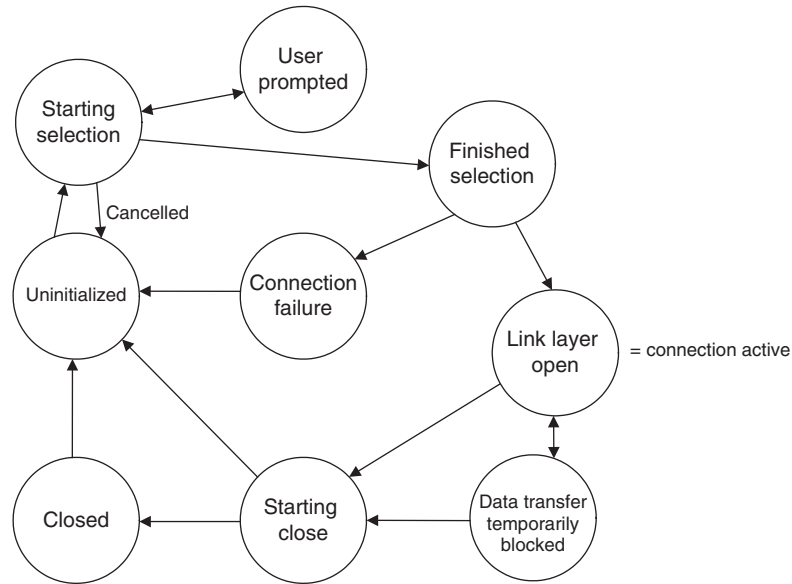


Figure 6.8 Connection progress stages

- **KStartingSelection** – the IAP to be used for the network connection is being selected. This is where the user will be prompted if necessary.
- **KFinishedSelection** – the IAP has been selected, the network connection can be attempted using the parameters configured in the IAP settings. Note that in some cases you may see the selection stages repeated many times if the system is trying alternate connections (for example, if the first connection cannot be made and the system moves to the second connection preference).
- **KLinkLayerOpen** – the connection has been successfully activated and, in most cases, an IP address has been allocated. This is the first stage where it is possible to send IP traffic. Situations where this stage is reached and an IP address might not have been allocated mainly involve DHCP over Ethernet, or Ethernet-like, networks.
- **KDataTransferTemporarilyBlocked** – data transfer on the connection is temporarily disabled. The reasons for this will be dependent on the bearer, but this could happen if, for example, a GPRS context is suspended because the user temporarily moves out of network coverage. Not all GPRS connections will send this progress stage, as it depends on device specific adaptation code to generate it, so you should not rely on receiving it. However, if it is sent, you should see **KLinkLayerOpen** again once data transfer is possible again.

- `KConnectionFailure` – the connection attempt has failed.
- `KConnectionStartingClose` – a disconnection has been initiated.
- `KLinkLayerClosed` – the connection has been terminated.

Applications subscribe for progress notifications like this:

```
TNifProgressBuf iProgBuf; // member of class
...
iConn.ProgressNotification(iProgBuf, iStatus);
```

This will complete when a progress change has occurred, and the `TNifProgress` will contain the latest progress value. `TNifProgressBuf` contains two parameters:

- `iStage` – the stage of the connection. One of the values defined above, or a bearer specific stage. Applications should be prepared to process, or at least ignore, unknown stages because they may not always be aware of which bearer is being used.
- `iError` – the error code at this stage. If the connection stage is successful, this will be `KErrNone`, otherwise it will be set to the reason why the stage has failed. An error is always followed by a return to the uninitialized state, possibly passing through a series of other stages on the way.

If your application is only interested in one particular status, such as an indication that the connection has been successfully activated, the required status can be included as an additional parameter. This will complete once the requested stage has been reached.

```
iConn.ProgressNotification(iProgBuf, iStatus, KLinkLayerOpen);
```

Because these are asynchronous notifications, and stage changes can occur rapidly during connection establishment and termination, applications should not assume that all stage changes will be retrieved by the application. Stages may be missed if the application has not re-issued a new notification before the stage change occurs and this means that the error notifications may also be missed. If an error occurs, the stage will eventually return to the uninitialized stage but the error is stored so that the application can always retrieve the reason for the failure/disconnection using `LastProgressError()`.

Because stages can be missed, if the application really wants to know the current stage, it should use the synchronous `Progress()` method to

retrieve the current progress information. A typical `RunL()` for detecting the error that has caused a connection to be terminated could look like this:

```
void CProgressTracker::RunL()
{
    if(iStatus.Int()==KErrNone)
    {
        // Repost the notification immediately to minimise chance
        // of missing new notifications
        iConn.ProgressNotification(iProgBuf, iStatus);

        if(iProgBuf().iStage==KConnectionUninitialised)
        {
            iConn.LastProgressError(iProgBuf());
            err = iProgBuf().iError;
            // now do something with err!
        }
        SetActive();
    }
    else
    {
        // some problem getting progress notifications
    }
}
```

6.5.2 Active Connection Information

`RConnection` provides the ability for applications to retrieve different types of information about an active connection.

Configuration information

Applications are able to read configuration information for the connection they are currently using. This saves the application from needing to use the `CommsDat` interface described in section 6.3.2 to retrieve the information, as this can all be done using `RConnection`. The settings are read in the form “<table name>\<field name>” (and there are constants defined for each of the tables and fields defined in `cdbcols.h`). For example, the IAP settings are stored in the IAP table, so to get the IAP ID (an integer), use “IAP\ID”.

```
_LIT(KIapTableIdField, "IAP\ID");
TInt iapId = 0;
err = iConn.GetIntSetting(KIapTableIdField, iapId);
```

Or to read the IAP name:

```
_LIT(KIapTableNameField, "IAP\Name");
TBuf<KCommsDbSvrRealMaxFieldLength> iapName;
iConn.GetDesSetting(KIapTableNameField, iapName);
```

6.6 Quality of Service

Some applications may have specific requirements for data they send and receive in terms of latency or bandwidth. Applications can specify these requirements by setting QoS parameters to ensure the correct behavior for the application.

Different network technologies implement QoS in different ways – at this time, only 3G packet data networks have a QoS implementation in Symbian OS. Therefore this section is only currently applicable to applications using 3G packet data connections.

6.6.1 3G QoS

3G QoS is defined in 3GPP 23.107 (R99). As is usual for QoS, it allows different services on a device to request different behaviors of the network, and the network operators can allocate different resources (bandwidth etc.) depending on the requirements of the active services. Examples of this are where an IP telephony application requires low latency (if the packets arrive too late to be played they will be thrown away and sound quality will decrease, or suffer from dropouts), whereas an music or video download requires high bandwidth. On 3G data networks, this is negotiated with the network via a secondary PDP context. As discussed at the start of this chapter, a secondary PDP context is a PDP context carrying some subset of the IP traffic for this connection, which has a specific QoS profile. Secondary PDP contexts do not have their own IP address, the IP address of the network connection is shared between the primary and secondary contexts (so the same IAP will be used for the primary and secondary context, and obviously they will connect to the same network).

The main concept of 3G QoS is the traffic class – there are four traffic classes, as shown in Figure 6.9.

There are four main characteristics to consider when thinking about QoS:

Conversational	Guaranteed Low delay Real time
Streaming	Guaranteed delay (but not necessarily low delay)
Interactive	Low delay (but not guaranteed error free)
Background	No guarantee on error free delay

Figure 6.9 3G traffic classes

- Bandwidth – this is the rate at which the connection can transfer data. Obviously some services want as much bandwidth as possible, for example, downloading, whereas others have fairly minimal requirements, for example, VoIP.
- Latency – this is the delay through the network between two endpoints. For ‘real-time’ services where some form of conversation is involved it is important that this be kept low enough that it doesn’t hamper communications – for voice networks a delay of less than 150 ms is often unnoticeable.
- Jitter – this is the variation in delay through the network, and is normally smoothed out at the receiver with the use of a jitter buffer. However, use of a jitter buffer increases the latency, so for some applications (voice again being a prime example) it is desirable to keep the size of the jitter buffer small.
- Packet loss – this is obviously the number or percentage of packets that are dropped during transfer from one endpoint to another.

6.6.2 Introducing RSubConnection

A *subconnection* in Symbian OS represents a channel as part of a network connection that has specific parameters associated with it. In the case of the 3G packet data network, it represents a secondary PDP context via the RSubConnection API shown in Figure 6.10. These behave in a similar way to RConnection but with the ability to also request QoS parameters for the connection. Each RConnection object has an implicit subconnection associated with it, which is automatically assigned the default QoS parameters (as configured by the network operator or phone manufacturer). This is likely to be best-effort bandwidth, with no other guarantees. Since the IP address is shared between the primary and secondary contexts, an RSubConnection uses the same network interface and IAP as its parent RConnection.

The QoS parameters are negotiated with the network – the application can provide two sets of parameters, the requested set and the minimum acceptable. The network will then either allocate the QoS in the range of the requested and minimum acceptable, or it will reject the request. This is done asynchronously, so the application must determine whether the QoS has been accepted using the event notification methods described a little later. But first, the configuration of the QoS parameters is discussed.

To create an explicit subconnection, there must first be a primary PDP context active via the parent RConnection. Then the subconnection can be opened:

```
RSubConnection iSubConn; // class member
...
err = iSubConn.Open(iSockServ, ECreateNew, iConn);
```

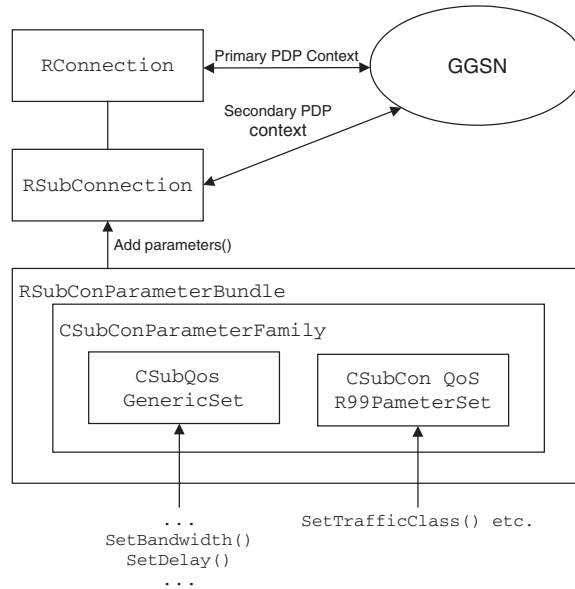


Figure 6.10 Setting QoS parameters on a subconnection

This associates the subconnection with the active network connection and an existing IP address. The application then needs to set the QoS parameters it requires from the network. This is done by creating an RSubConParameterBundle and filling it with the correct parameters. There are different sets of parameters that can be used – a generic set, i.e. one that is not specific to the bearer being used, or a bearer specific set (in this case, 3G). The different parameters are added to the RSubConParameterBundle using a container CSubConParameterFamily, which in turn has the generic and 3G QoS parameters added to it.

The generic QoS parameters are represented by the CSubConQosGenericParamSet and the 3G QoS parameters are represented by the CSubConQosR99ParamSet.

Generic QoS parameters

The generic QoS parameters allow the application to request the following:

Parameter	Description
Bandwidth	Bandwidth the client requires
Maximum burst size	Maximum size of a burst of data the client can handle

Parameter	Description
Average packet size	Average packet size required (e.g. codec use)
Maximum packet size	Maximum packet size the client can handle
Delay	Acceptable delay/latency
Delay variation	Acceptable variation in delay (also known as jitter)
Priority	Relative priority the client expects to give this channel compared to its other channels
Header mode	Specify whether the header size should be calculated by the QoS module or specified by the client. Default is to let the QoS module calculate it
Name	Identity of a ‘well-known’ set of QoS parameters

So to create a generic set of QoS parameters:

```
RSubConParameterBundle paramBundle;
// Create the container object
CSubConParameterFamily* qosFamily =
    CSubConParameterFamily::NewL(paramBundle, KSubConQoSFamily);

// Create the minimum accepted and request parameter sets
CSubConQosGenericParamSet* genericMinParams =
    CSubConQosGenericParamSet::NewL(qosFamily,
    CSubConParameterFamily::EAcceptable);
genericParams->SetDownlinkBandwidth(128);
genericParams->SetDownLinkDelay(64);

CSubConQosGenericParamSet* genericRequestedParams =
    CSubConQosGenericParamSet::NewL(qosFamily,
    CSubConParameterFamily::ERequested);
genericParams->SetDownlinkBandwidth(128);
genericParams->SetDownLinkDelay(64);
```

The RSubConParameterBundle now takes ownership of the CSub-ConParameterFamily so the client does not need to delete it later.

If only the generic set of parameters is requested, the system will make a best-effort mapping to the 3G QoS settings. However, in some cases,

the application may be better placed to set the 3G QoS parameters itself, although this will obviously mean that further changes are necessary as further QoS-aware bearers are introduced in the future.

3G QoS parameters

The 3G QoS parameters are handled in a very similar way to the generic set, but using the `CSubConQoS99ParamSet`. This allows all of the QoS parameters as defined in 3GPP 23.107 to be configured by the application. The parameters are mapped directly onto the QoS parameters defined in the Symbian telephony API, `RPacketQoS`, therefore see the documentation of `RPacketQoS` for more details.

```
// Create the minimum accepted and requested 3G parameter sets
CSubConQoS99ParamSet* minParams = CSubConQoS99ParamSet::NewL(qosFamily,
    CSubConParameterFamily::EAcceptable);
minParams->SetTrafficClass (ESTreaming);
minParams->SetMaxBitrateDownlink (256);

CSubConQoS99ParamSet* requestedParams =
    CSubConQoS99ParamSet::NewL(qosFamily,
    CSubConParameterFamily::ERequested);
requestedParams->SetTrafficClass (ESTreaming);
requestedParams->SetMaxBitrateDownlink(128);
```

These parameters can now be added to the subconnection:

```
TInt err = iSubconn.SetParameters(paramBundle);
```

It is also possible to use the `SetGenericSetL()` and `AddExtensionSetL()` methods to add the parameter sets to the `CSubConParameterFamily` instead of using the version of `CSubConQoSGenericParamSet::NewL()` which does it implicitly.

```
RSubConParameterBundle paramBundle;
// Create the container object
CSubConParameterFamily* qosFamily =
    CSubConParameterFamily::NewL(paramBundle, KSubConQoSFamily);
CSubConQoSGenericParamSet* genericMinParams =
    CSubConQoSGenericParamSet::NewL();
genericMinParams-> SetDownlinkBandwidth(128); // WHAT UNITS ARE THESE IN?
genericParams-> SetDownLinkDelay(64); // WHAT UNITS ARE THESE IN?
qosFamily.SetGenericSetL(genericMinParams,
    CSubConParameterFamily::ERequested);
```

Event notifications

The application must listen for QoS event notifications to determine whether the QoS has been accepted by the network before continuing with data transfer on the subconnection:

```
iSubconn.EventNotification(iNotifBuf,ETrue,iStatus);
```

This completes when one of the following events occurs:

```
CSubConGenEventParamsGranted
```

QoS negotiation with the network has been completed successfully. The application can then retrieve the negotiated QoS parameters from the CSubConParameterFamily, which will be somewhere between the requested and minimum acceptable.

```
CSubConGenericParameterSet* negotiatedQoS =  
    qosFamily.GetGenericSet(CSubConParameterFamily::EGranted);
```

or

```
CSubConQoSr99ParamSet * negotiatedQoS =  
    qosFamily.GetExtensionSet(KSubConQoSr99ParamsType ,  
        CSubConParameterFamily::EGranted);  
  
CSubConGenEventParamsRejected
```

QoS negotiation with the network has failed – the network would not allocate the QoS requested. The secondary PDP context has not been activated. The reason for the failure can be determined using the Error() method.

```
CSubConGenEventParamsChanged
```

The QoS has been renegotiated due to some event on the network. This event is triggered by the network, not by a request sent by the application.

```
CSubConGenEventSubConDown
```

The subconnection has been lost, that is, the secondary PDP context has been terminated. This could be initiated from the device closing the connection, or by the network if an error occurs. The actual error can be determined using the Error() method.

```
CSubConGenEventDataClientJoined
```

A socket has been added to the subconnection. See ‘Adding sockets to a subconnection’ below.

```
CSubConGenEventDataClientLeft
```

A socket has been removed from the subconnection. See ‘Adding sockets to a subconnection’ below.

If the network does not allocate the requested QoS settings, then the necessary channel characteristics cannot be guaranteed. The application developer then needs to make a decision on whether to continue using the parent `RConnection` object, that is, using the default QoS parameters. In this case, the application will have to handle any problems caused by the incorrect QoS settings, such as buffering data if it isn’t received quickly enough.

Adding sockets to a subconnection

Once the subconnection is active, data can be sent over it by attaching sockets to it. All data transferred using the attached sockets will then use the secondary PDP context. There are two possible ways of doing this:

1. Open the socket using `RSubConnection` instead of `RConnection`:

```
TInt err = iSock.Open(iSockServ, KAfInet, KSockStream, KProtocolInetTcp,
    iSubconn);
```

2. Adding an existing socket to the subconnection (in this case, the subconnection must be related to the parent connection used to open the socket originally):

```
iSubconn.Add(iSock, iStatus);
```

Note that any errors relating to the network/secondary PDP context/QoS will be returned via the event notifications. Other errors returned will be programming or system errors relating only to the immediate operation. This is handled asynchronously since interaction with the network is required (the device has to give information to the network to allow it to route that socket’s data over the secondary PDP context). When the socket has been successfully added (or removed using `Remove()`), the `CSubConGenEventDataClientJoining` (or `CSubConGenEventDataClientLeaving`) events described earlier are notified.

6.7 Summary

In this chapter, we have learnt how to:

- select a specific connection for our application to use, or get the system to prompt the user on our behalf

- monitor the progress of a connection as it is started and stopped so we can provide feedback to the user
- perform DNS lookups to map DNS names to IP addresses
- use TCP and UDP sockets
- use TLS over a TCP socket
- configure QoS on 3G networks.

7

Telephony in Symbian OS

The telephony subsystem is the interface to the cellular network functionality in the phone. As we mentioned in Chapter 2, this is sourced by the phone manufacturer and typically runs on a different OS and different processor core to Symbian OS. In practice, it is the gateway to the services provided by your network operator – both voice and data.

The telephony subsystem provides support for W-CDMA (3GPP R4 and R5), GSM circuit-switched voice and data (CSD and EDGE CSD) and packet-switched data (GPRS and EDGE GPRS). In addition, there is also support for CDMA circuit-switched voice and data, and packet-switched data (IS-95 and 1xRTT), although there are no phones supporting this standard on the market. Access to the SIM, RUIM or UICC is also provided.

The main purpose of the telephony APIs are to provide access to the cellular network. The type of access provided can be divided into two categories – data-centric or voice-centric.

Data-centric applications will typically access the phone network through higher level APIs rather than those provided by ETel – in particular, they will use the ESOCK APIs such as `RConnection` to create connections to the cellular network, in the same way as they would for other types of network, then `RSocket` to transfer data over that connection. This abstracts the details of creating a connection over the cellular network away from the application, as the interface to ETel is handled by the networking subsystem, and the connection made using information from the communications database (Commsdat).

Voice-centric applications are typically the ones that handle voice calls, including placing an outgoing call or answering an incoming one. They may also make the use of Supplementary Services such as call barring and forwarding. Voice-centric applications use the telephony subsystem directly, rather than using a higher-level API.

Telephony APIs also provide access to information contained on the SIM card, such as the IMSI number; on the device, such as the IMEI; or from the network, such as the ID of the cell in which we are located.

In order to provide access to this functionality, a restricted set of APIs is offered via a component called ETel ISV, which is also known as ‘ETel 3rd Party’. We will explain how to use this API and provide an example application, as well as documenting the precautions that must be taken when using these APIs, along with their limitations.

There are other components that form part of the telephony subsystem that we will omit from this chapter, as they do not expose APIs for developers – in particular, the phonebook synchronizer, which is only distributed as part of UIQ.

VOIP Calls cannot be established using Etel ISV please refer to the SIP/RTP chapters.

7.1 Overview

Figure 7.1 shows a high-level overview of the telephony subsystem and its major functional blocks, as well as the main interfaces between components.

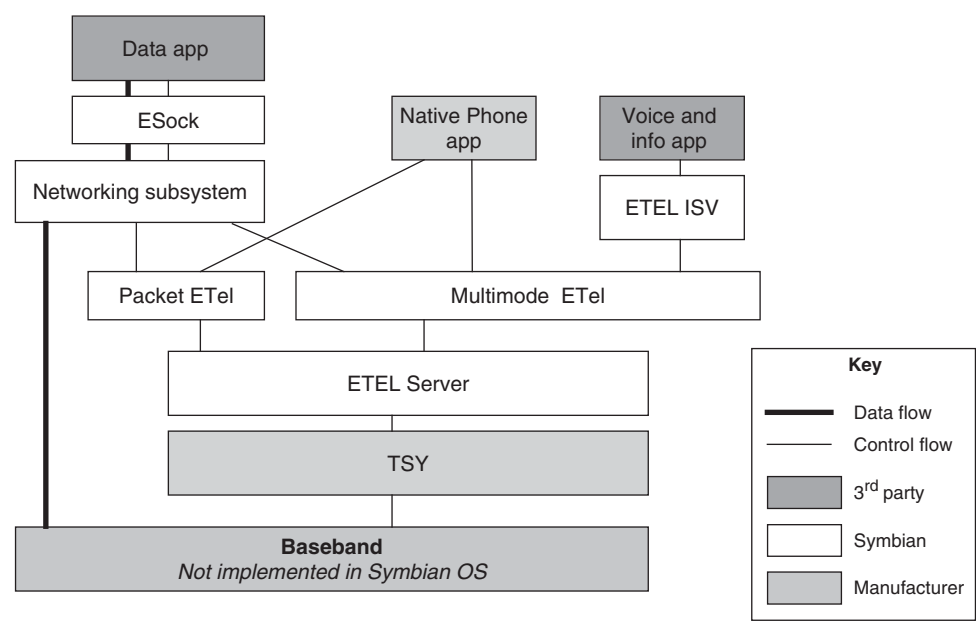


Figure 7.1 Telephony subsystem

7.2 Using the ETel ISV API

As part of the example applications, we will show two major areas for which you may wish to use the ETel ISV API.

The first is making outgoing phone calls. This allows you to place outgoing calls from your application – for example if you are providing some sort of directory application, such as Yellow Pages, and wanted to allow users to place a call to a company directly from their listing. This is particularly effective in cases where the calls are made on an outgoing basis only, as no integration with the built-in contacts database is necessary. It also has the benefit of allowing many numbers to be stored in an application's own local database rather than cluttering up the user's main contacts database. However, in the case where an incoming call is received, this solution won't be as user friendly because the phone number-to-name matching takes place against the user's main contacts database, so will not match entries in the application's local database. The UI does not need to handle the call if the application does not run in the foreground the native phone app will handle it.

The second main area of interest is detecting incoming calls. There may be a several motivations for this – we'll cover just two. If we are creating an application that uses audible output, it is useful to detect incoming calls so that we can suppress our audio output whilst the user is on a call. It is also useful to detect when the call has been terminated to allow us to re-enable our audio output. The other reason for detecting incoming calls is to allow us to implement such applications as video ringtone engines, where we may wish to play a video clip when an incoming call is detected.

Other interesting things that can be done using the ETel ISV API include:

- Monitoring the current cell ID, for example to use as a location in games – some games could require you to return to a specific location in order to complete a task – bringing some element of real-life space into the game.
- Checking whether the user is roaming or not, and modifying certain aspects of application behavior – for example, reducing the rate at which scheduled information retrieval is performed using the network to reduce costs to the user.
- Using the IMEI or IMSI to create a very basic 'locking' scheme for your application to prevent casual copying – we discuss this in section 7.2.2.

7.2.1 Example Application

The functionality of the ETel ISV API is exposed through `CTelephony`. Among other functionality, this provides access to the following information:

- The Subscriber ID, also known as IMSI, which uniquely identifies your ICC (aka SIM card).
- The Phone ID, which comprises the manufacturer, model and a serial number (a.k.a. IMEI).
- The network registration status.

The example application will retrieve all this information, and also requests to be notified of changes to the network registration status.

Implementation

As is traditional in Symbian OS development, the application has been split in two – an engine and a UI, to allow reuse of the engine with multiple UIs. Figure 7.2 shows the high level design of the application.

The engine is a container for the active objects that are used to call and wait for the completion of the `CTelephony` asynchronous methods.

`CGetter` retrieves information from the phone using the getter methods on `CTelephony`. We use two of them: one for the subscriber ID contained in the `CTelephony::TSubscriberIdV1` package; and one for the IMEI contained in the `CTelephony::TPhoneIdV1` package. When started, they will place a request on `CTelephony`. Their `RunL()` notifies the engine of the completion.

Their functionality has also been extended to create `CObservers`, which also notify the application of the change in this information – our application uses one of these to retrieve the network registration status.

Dial will also take a `TCallparamV1` argument, which specifies whether the caller identity (your telephone number) should be sent to the remote user.

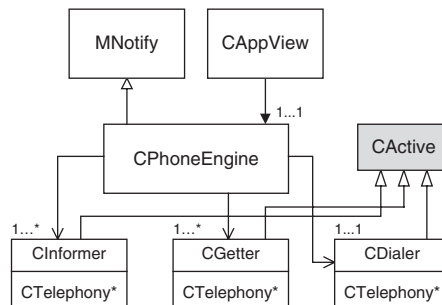


Figure 7.2 High-level design of the telephony application

Please note that on UIQ you can use DNL (Direct Navigational Link), which will call into the phone app to dial a call. I would recommend this solution to using CTelephony because of its simplicity and its good integration with the phone application.

The requests are made by calling the member functions of the application's CPhoneEngine, which dispatches the request to the appropriate active object. It is also the engine's responsibility to handle the processing of the data such as the matching of the unlock code coming from the UI to the unlock code generated using the IMEI of the device. Not only does it make it safe but it reduces the need to make UI variants modifications.

The UI takes commands from the user and displays their results.

The notifications from the engine are made via an abstract class MNotify that the UI implements.

```
class MNotify //interface to the UI
{
public:
    virtual void DialCompleteL(TInt aError)=0;
    virtual void VerifyIMSICompleteL(TInt aError, TDesC& aIMSI)=0;
    virtual void VerifyIMEICompleteL(TInt aError, TDesC& aIMSI)=0;
    virtual void UpdateNetworkRegistrationStatusL(TInt aError,
TDesC& aNetworkRegistrationStatus)=0;
}
```

As with many Symbian OS services, this engine requires the thread using it to have an active scheduler installed and running.

It is possible to have several client applications running at the same time. Each application should instantiate a CTelephony object – the Multimode ETel API deals with having several clients making requests, as well as completing notifications to several clients.

Notifications

The telephony subsystem has been designed to cope with multiple notifications completing to several clients. However, events can occur whilst a client is processing a previous notification. In order to avoid missing such events, any processing performed in one of the MNotify callback methods should be kept as short as possible, to allow the engine to make the next request for notification.

This can often be achieved very easily by simply taking a copy of the information, storing it in an CActive-derived class that you've designed for this purpose, then immediately completing that active object.

7.2.2 Using IMSI or IMEI to Provide Simple Application Locking

To prevent casual copying of your application, you may want to implement a basic scheme in order to 'lock' your application to a single handset or ICC (aka SIM card). This can be done by using the IMEI (or

IMSI) number of the handset (or ICC), in conjunction with some form of hashing scheme to generate ‘unlock codes’ for specific handsets.

Note that this only provides a basic level of protection to your application. Determined attackers can still modify your application to bypass the checking code. If you are intending to market a high-value application, or have a specific requirement to prevent copying, then we suggest you take a look at more advanced technologies, such as DRM-protected SIS files, for distribution.

To implement a locking scheme, you will need to get the user’s IMEI or IMSI, perform some form of hashing function on it, then send them the hash to enter into the application on their phone.

Here’s a more detailed suggestion on how to do this:

(a) Ask the user who purchased the product to provide you with its IMEI or IMSI number. You can do this automatically, if the application is already installed on the phone by using `CTelephony::GetPhoneId()` to retrieve the IMEI and `CTelephony::GetSubscriberId()` to retrieve the IMSI.

Alternately, if the application is not already on the phone (perhaps you’re generating the unlock code as part of selling the application from a website) then you’ll need to use the manual process. The IMEI number can be retrieved by entering `*#06#` into the telephony application. The sequence to dial for the IMSI differs for each network – you may have to refer users to their network operator to obtain this information, or provided instructions for popular networks.

(b) Generate an unlock code and send it to the user. You need to create your algorithm to do this. It needs to be implemented in your application for the verifying process and also on another machine so that you can generate the unlock code.

The algorithm could be a very simple function – for the purposes of this example, let’s just add the IMEI number to a ‘secret’ number to generate an unlock code – obviously this isn’t very secure so you should use a better algorithm in practice!

So we do: `< IMEI > +132456789 =< unlock code >`. Now we have generated an unlock code, we need to send it to the user.

(c) User starts the application and enters the unlock code. How the user enters the unlock code depends again on how the application is being sold – we could display it on a web page and get the user to enter it manually, or we could combine it with the example for silently receiving SMS messages from Chapter 8 to automatically unlock the application on the user’s device (obviously we’d need some method for sending SMS messages, and their phone number in this case!).

When the application starts, it can retrieve the IMEI or IMSI using `CTelephony`, internally generate the unlock code and verify if it matches the unlock code entered by the user. If the two unlock codes match then the application can run normally.

Public key encrypted IMSIs or IMEIs can also be used in the authentication process of a connection to a remote service.

7.3 Restrictions and Considerations

7.3.1 Access to Telephony Functionality

The Symbian MultiMode ETel APIs offer full access to the TSY and therefore the baseband, and represent a very powerful set of APIs. They offer the ability to control voice and data calls, GPRS and 3 G data connections, as well as information stored on the device or on the network. Because of the sensitivity of the information exposed, and the potential to prevent the device operating correctly, these APIs were not made available to developers before v9.1, as there was no method for policing access to them.

In order to offer generally available APIs for telephony services a new interface was created, designed with an easy-to-use format, to allow access to a controlled subset of the MultiMode ETel APIs. This was released in Symbian OS v7.0 and was originally named ETel 3rd Party. It concentrated mainly on data call functionality.

A newer version was released in Symbian OS v8.0, renamed ETel ISV, and offered voice call support and access to more information about the current state of telephony on the device. In later versions of ETel ISV the need for data calls was deprecated in favour of clients using more obvious and convenient data transfer methods such as GPRS.

With the advent of platform security in v9.1, it was possible to restrict access to sensitive parts of the MultiMode ETel APIs. As a result, they are now published in the UI platform SDKs. However, the ETel ISV API remains, both for backwards compatibility, and because it is much easier to use than the extreme feature-rich, but complicated, MultiMode ETel APIs.

If you do wish to use the MultiMode ETel APIs directly, please be aware that each method can be policed by different capabilities. Some calls may not require any capabilities at all, for example `GetFlightMode()`. However, many calls require at least one of the following three – `ReadUserData`, `ReadDeviceData` or `NetworkServices`. Please see the Symbian OS Library for more details.

In Figure 7.3 we show the capabilities required to use the functionality on the ETel ISV API.

Initially in v9.1, some functionality was protected by higher capabilities than shown in Figure 7.3. However, in order to allow some of the use

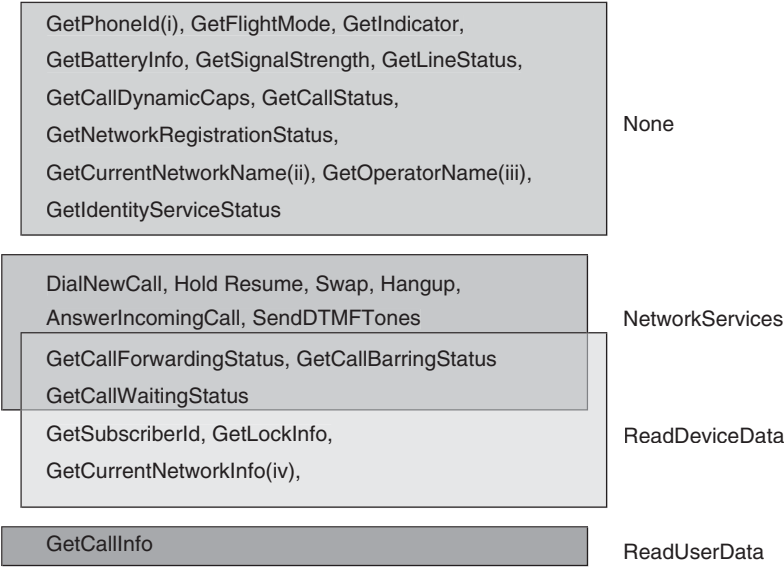


Figure 7.3 Capabilities required for different operations on ETel ISV API

GetPhoneId	ReadDeviceData
GetCurrentNetworkName	ReadUserData
GetOperatorName	Location
GetCurrentNetworkInfo	ReadDeviceData ReadUserData Location

Figure 7.4 Capabilities required for certain operations in early v9.1-based devices

cases described in this chapter, the capabilities required to access some information were reduced. However, some early v9.1 phones will police access using the old capabilities as described in Figure 7.4.

As some phones have already been shipped with the old capabilities it may be necessary to ensure you have the old set of capabilities in order for your application to work on all v9.1-based devices. However, the other option is to ask users to upgrade their firmware to a more recent release which will only perform policing based on the new requirements.

This concerns mainly S60 phones as very few UIQ phones would have been shipped with the old capabilities.

7.3.2 Dialling a Call

In order to dial a call, a number string needs to be entered into the phone and passed into the `Dial()` method.

Emergency calls

Please be aware that if you intend to write any code that affects the ability of the device to make phone calls then you need to be exceptionally careful to ensure emergency calls will still always succeed. `CTelephony` is designed to allow an application to manage its own calls without interfering with calls from other applications, so by using `CTelephony` to access telephony functionality you fulfil this requirement.

7.3.3 Answering a Call

All Symbian OS-based phones have a built-in phone application that enables the user to make, receive and manage voice, and possibly video, calls.

When an incoming call is received the built-in phone application will get an incoming call notification, and will perform various actions such as displaying the caller's number, or name if they exist in the contacts database, present options to answer or reject the call, and, depending on the current settings, play the ringtone. Most phones will incorporate some form of audio policy manager that ensures the ringtone belonging to the built-in phone application will override any other audio output. This means that any system that attempts to override the built-in ringtone, such as a video ringtone application will need to interface with UI platform-specific APIs in order to disable the built-in ringtone. On S60, this would be the Profile Engine Wrapper API, documented in the S60 3rd edition Feature Pack 1 SDK, and available starting from that release. On UIQ, this is manufacturer specific: Sony-Ericsson will mix your sound with a beeping alert, Motorola will not allow built in ring tones to be disabled at all.

7.3.4 Notifications

If you wish to get notified of an event change you can use `CTelephony::NotifyChange()`, passing as one argument the request status and the other the correctly packaged notification object descriptor that will contain the result on completion.

You can have as many different outstanding notifications as you have active objects handling them. `ETel 3rd Party` can manage several outstanding notifications concurrently provided they are for different events. Using our classic design for `ETel 3rd party` asynchronous requests, we need an Active Object for each type of notification requested and handle each `RunL` with the appropriate type.

7.3.5 Video Calls

Video calls are not supported by the `ETel ISV` APIs. On UIQ you can use `DNL` (Direct Navigational Link) which will call into the phone application,

from the `QPhoneAppExternalInterface.h`. This can also be used for placing voice calls. S60 has no public interface for creating video calls.

7.4 Summary

In this chapter we have learnt how to:

- access the telephony subsystem through the interfaces provided to developers
- place outgoing calls
- listen for, and answer if required, incoming calls
- monitor the network status, including current cell ID and network
- retrieve information about the handset and ICC in use – in particular the IMEI and IMSI.

Section III

High-level Technology and Frameworks

8

Receiving Messages

Symbian OS-based phones typically provide support for several different messaging technologies – built-in applications allow users to send and receive email, SMS and MMS messages. It is possible for third-party applications to use and extend this messaging functionality in order to add new and innovative features. Some possible messaging-aware applications might include:

- an email management application that filters and arranges received email messages
- a subscription service that receives a news report over MMS and renders it using a custom application
- a multiplayer game that uses a messaging transport (such as SMS) to send moves between handsets
- a video SMS ringtone application which plays a movie clip every time an SMS message is received.

This chapter explains how to use the messaging functionality in Symbian OS-based devices. It introduces the key messaging concepts in Symbian OS and covers the following:

- The Symbian OS message store
- The message server and the associated client APIs
- Listing messages in the message store
- Extracting summary information from received messages (e.g., the sender and subject)

- Waiting for new received email and SMS messages to appear in the inbox
- Intercepting SMS messages before they reach the inbox.

Using the SendAs functionality, which builds on top of the messaging framework, is covered in Chapter 9. Chapter 9 also covers extending the messaging framework with a new set of Message Type Modules (MTMs) to add a new type of transport for SendAs.

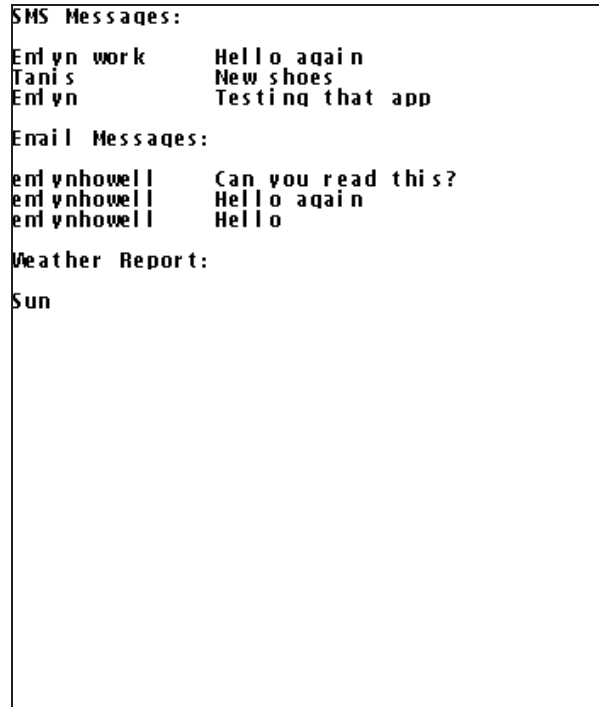


Figure 8.1 The summary screen application in action

8.1 Example Application – Summary Screen

The summary screen application (as seen in Figure 8.1) is used to illustrate the techniques discussed in this chapter, it provides the following functionality.

- Displays the details of the most recent email¹ and SMS messages, i.e., the sender and subject text.

¹ In order to keep the example code simple, we only check the first POP3 account, if present.

- Interprets special ‘Weather Report’ SMS messages and displays the appropriate text. These messages are intercepted before they reach the messaging inbox.

Figure 8.2 illustrates the class structure of the summary screen example application.

The following lists the responsibilities of the main classes in the summary screen application:

- **CSummaryScreenUI** – implements a very simple text-based UI. This class does not use any messaging APIs directly.
- **CMessageSummaryEngine** – the main engine class. It creates the appropriate messaging client objects but does not actually generate the message summaries.
- **CMessageSummaryGenerator** – generates and stores the message summaries for a specified message type. There are two instances of this class, one for SMS and one for email. The same CMessageSummaryGenerator class can be used for both SMS and email because the Symbian OS APIs for retrieving basic message attributes are the same for all message types.
- **CWeatherReportWatcher** – waits for special weather report text messages. It accomplishes this by registering for notification of any text message starting with the prefix “Weather:”. When a weather report

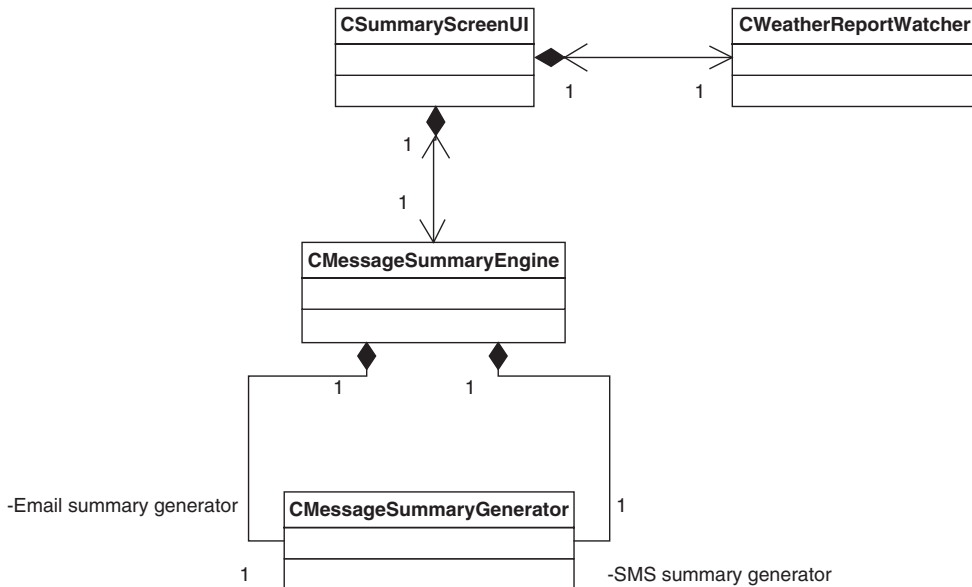


Figure 8.2 Structure of the summary screen example application

is received, it is decoded and the UI is notified via a callback. Note that this class will intercept the SMS message before it reaches the message server and the inbox, in order to prevent a user notification being triggered. Therefore all message server APIs are bypassed for weather reports.

8.2 The Message Server

8.2.1 Message Server Architecture

Symbian OS includes support for a number of messaging-related technologies such as POP3, IMAP4 and SMTP email transports, and SMS. These messaging-related features are usually accessed via the message server. The message server provides the following functionality:

- Storing and retrieving messages using the message store.
- Allowing multiple client applications to receive, create and send messages.
- Notifying multiple client applications of changes to the message store, e.g., when a new message is received.

The message server can be extended to support new message types by implementing additional MTMs. A simple MTM example, which extends the SendAs service with a new transport type, is covered in Chapter 9.

The remainder of this chapter concentrates on how client applications can access messaging functionality using the built-in message types. The examples used in this chapter include SMS and POP3 email. However, the key concepts presented here also apply to other message types, such as MMS.

8.2.2 Message Server Session

The first messaging-related task that a client application must perform is to open a session to the message server. The messaging session allows a client application to examine the message store, create and send messages and to be notified of any messaging-related events (for example a received SMS message).

Example 8.1 shows the declaration of the messaging session.

```
class CMessageSummaryEngine : public CActive, public MMsvSessionObserver
{
public:
    ...
```

```

void HandleSessionEventL(TMsvSessionEvent aEvent, TAny *aArg1, TAny
    *aArg2, TAny *aArg3);
...
private:
    ...
    // Symbian OS messaging classes
    CMsvSession* iMessagingSession;
};

```

Example 8.1 Part of the class declaration for the messaging summary engine, showing the message server interfaces

The message server session is encapsulated by the `CMsvSession` class. It is highly recommended that only one `CMsvSession` object is instantiated per application because additional sessions will have an impact on system performance and RAM usage.

Classes that own a `CMsvSession` will usually implement the `MMsvSessionObserver` interface. The `MMsvSessionObserver` interface (specifically `HandleSessionEventL()`) is called by the message server to notify the client about message server events, for example that a new message has been created (see Figure 8.3).

Example 8.2 shows how the `CMsvSession` is created:

```

// Start the summary operation
void CMessageSummaryEngine::StartL()
{
    // Connect to the Message Server asynchronously
    iState = EConnectingToMessageServer;
    iMessagingSession = CMsvSession::OpenAsyncL(*this);
    // The message server will call HandleSessionEventL when the session is
    // connected.
}

```

Example 8.2 Creating the `CMsvSession`

In the above example the `CMsvSession` is created by calling `OpenAsyncL()`. The 'this' parameter is passed in so that `CMessageSummaryEngine` is notified about future message server events

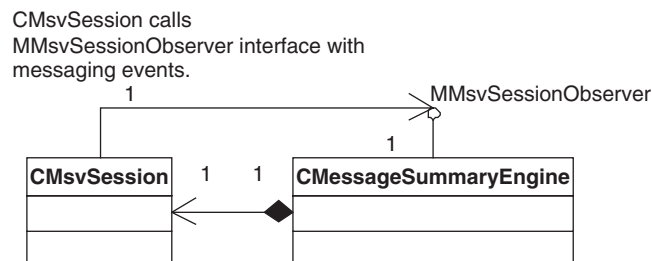


Figure 8.3 Interfacing with the messaging session classes

via its `HandleSessionEventL()` function. This is possible because `CMessageSummaryEngine` is derived from `MMsvSessionObserver`.

The client application does not require any platform security capabilities in order to open a `CMsvSession`. However, subsequent calls to the message client APIs may require specific capabilities depending on the operation type, for example, putting an email message in the outbox requires `ReadUserData`, `WriteUserData` and `NetworkServices`.

It is possible to create a messaging session either synchronously or asynchronously. This example uses the asynchronous version. There are several advantages to using asynchronous function calls for potentially long-running operations such as the creation of the messaging session:

1. The application remains responsive to user input.
2. Other active objects in the application thread may continue to run, performing other tasks while the messaging session is being created.

The message server signals that the messaging session has been created by calling the `HandleSessionEventL` function. The `CMsvSession` class cannot be used until this function has been called.

```
void CMessageSummaryEngine::HandleSessionEventL(TMsvSessionEvent aEvent,
    TAny *aArg1, TAny *aArg2, TAny* /* aArg3 */)
// This function is called by the message server for
// every message server event,
// e.g. when a new message is received or created.
{
    switch (aEvent)
    {
        case MMsvSessionObserver::EMsvServerReady:
            // The session is now connected to the message server
            if (iState == EConnectingToMessageServer)
            {
                // Create the summary generators for each message type
                CreateSummaryGeneratorsL();

                // If we were waiting for connection to the message server then
                move // on to the next state, update the message summaries.
                UpdateNextSummary();
            }
            break;
            ...
    }
}
```

Example 8.3 Waiting for the message server connect to complete

The `aEvent` parameter is used to determine the type of the notification. The code in example 8.3 checks the `aEvent` parameter to determine if the messaging session is initialized and then initiates the message summary generation.

8.3 The Message Store

8.3.1 Message Store Structure

The message store uses a tree structure to store messages. Every node in the tree structure is represented by an entry. Each entry can be one of four different types:

- **Folder entry** – used to group messages in much the same way as a directory in the file system is used to group files. Examples of folder entries include the inbox and the outbox.
- **Message entry** – used to store the contents of a particular message. One message entry is created for each message in the store. Note that some message types have child attachment entries which are used to store attachment data.
- **Service entry** – represents an external messaging account, for example a POP3 mailbox or an MMS account.

In versions of Symbian OS prior to v9.0 these service entries held the message account settings, for example POP3 server address, user name and password. However, in v9.0 onwards this information is stored in the Central Repository and is accessed via message-type specific APIs provided by the client MTM.

- **Attachment entry** – used to store the attachment or multimedia data for some message types.

Figure 8.4 shows an example message store structure, populated with assorted accounts and messages. It shows several SMS messages in the inbox and three different email accounts (two POP3 and one IMAP4).

Note that Figure 8.4 is a simplified representation of a typical message store. Each email message would usually have several corresponding attachment entries and additional folders would be present under the local service entry, for example a Drafts folder and a Sent folder.

Received push messages, for example SMS and MMS, are initially stored in the local inbox. A client application may later move the message entry to another location, for example the 'saved messages' folder.

Received POP3 and IMAP4 messages are stored under the appropriate service entry. The structure under this service entry corresponds to the message structure on the email server.

8.3.2 Message Entries

Just as the message store is broken up into entries, each individual entry also has a number of distinct components: some of these components

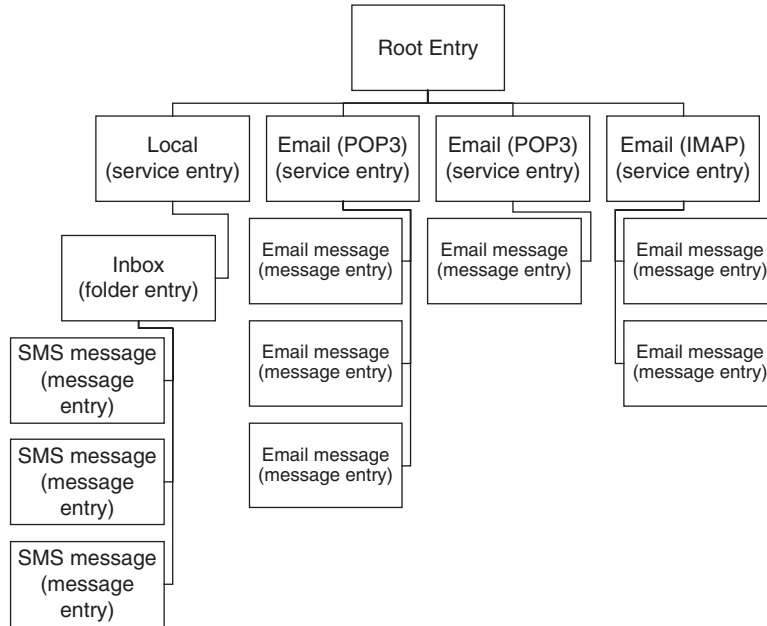


Figure 8.4 Simplified version of the contents of a typical message store

are generic to all message types and some are specific to a particular messaging technology. The structure of an individual message entry is shown in Figure 8.5.

We'll now look at the generic message attributes and message-type specific streams.

Generic message attributes

The exact format of each message entry depends on the message type and the state of the message. However, there are some attributes that are common to all message entries, for example:

- subject (or summary)
- date
- sender/recipient details.

These common attributes are accessed using the same APIs, regardless of the message type. This design means that it is possible to use the same source code to process the summary information for different message types. This is illustrated in the example summary screen application which uses the same class, `CMessageSummaryGenerator`, to generate

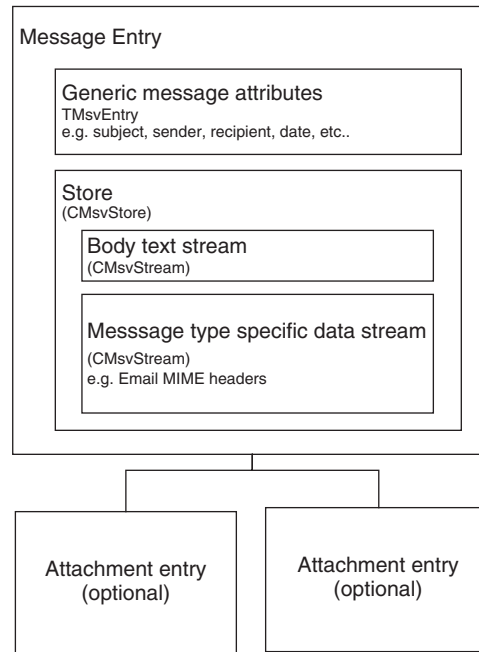


Figure 8.5 Structure of an individual message entry

the summary information for email and SMS messages. There is no code in this class that is specific to either SMS or email.

Some message types have a main text part – for an email (or MIME) message this would be the body text. Message-type specific APIs can be used to extract this body text, for example, `CImEmailMessage::GetBodyTextL()`. There are generic methods for extracting body text, but they do not handle character conversion for some message types, so it is better to use the message-type specific APIs if they are available.

Message-type specific data

In addition to the generic attributes, each message entry also has some information that is specific to the message type, for example email messages have MIME headers, and SMS messages have an associated service centre address. Each message entry has an associated store for these message-type specific attributes. Several message-type specific classes are provided for the purpose of extracting data from these stores. Examples of these message-type specific classes are:

- `CSmsHeader` for SMS messages
- `CImEmailMessage` for email messages.

A detailed exploration of these classes is beyond the scope of this book. However, documentation can be found in the Symbian OS Library.

Attachment entries

An email or MMS message entry may have associated attachment entries. Each attachment entry represents an attached file or media element. The attachment API should be used for accessing these attachments. An attachment manager object for a particular message can be accessed by calling `CMsvStore::AttachmentManagerL()`. Further information on this API and the `MMsvAttachmentManager` interface can be found in the Symbian OS Library. The `CMsvStore` object can be extracted from a `CMsvEntry` by calling `CMsvEntry::ReadStoreL()`. `CMsvStore` is discussed later in this chapter.

8.3.3 Message Entry APIs

CMsvEntry

If there is one key class in the messaging APIs, then `CMsvEntry` is it. However, care should be taken when using `CMsvEntry` as inappropriate use can seriously impact system performance and degrade the user experience-specific advice on usage can be found in section 8.5.3.

`CMsvEntry` is used for the following purposes:

- to navigate around the message store
- to enumerate the child entries, for example to retrieve a list of message entries in the inbox. Entries can be sorted by specified criteria
- to retrieve the store for the message-type specific information (e.g., email MIME headers)
- for modification of entries (e.g., change, copy, move and delete operations).

`CMsvEntry` can be thought of as a pointer to an entry in the message store. It may point to any type of entry, for example, a folder entry or a message entry. It is common for a messaging application to use a single `CMsvEntry` to point to a particular entry, to extract the required information, and then to set it to point at another entry.

```
class CMessageSummaryEngine : public CActive, public MMsvSessionObserver
{
    ...
    // Symbian OS messaging classes
    CMsvSession* iMessagingSession;
    CMsvEntry* iMessagingEntry;
    ...
};
```

```

void CMessageSummaryEngine::CreateSummaryGeneratorsL()
{
    // Select the ordering we wish to use when enumerating through messages.
    // The message summary application is only interested in the most recent
    // messages so EMsvSortByDateReverse is appropriate.
    TMsVSelectionOrdering ordering(KMsvNoGrouping, EMsvSortByDateReverse);

    // Create the CMsvEntry object that will be shared by all summary
    // generators
    // Note that it is only possible to create the CMsvEntry once the
    // message session is connected.
    iMessagingEntry = CMsvEntry::NewL(*iMessagingSession,
                                      KMsvGlobalInBoxIndexEntryId, ordering);

    iSmsInboxSummary = new (ELeave)
    CMessageSummaryGenerator(KMsvGlobalInBoxIndexEntryId, KUidMsgTypeSMS,
                            *iMessagingEntry);

    // Find the first POP3 mailbox
    TMsvId pop3MailboxId;
    if (FindPop3ServiceEntryL(pop3MailboxId))
    {
        iEmailInboxSummary = new (ELeave)
        CMessageSummaryGenerator(pop3MailboxId, KUidMsgTypePOP3,
                                *iMessagingEntry);
    }
}

```

Example 8.4 Creating a CMsvEntry

Example 8.4 shows how a CMsvEntry object is created. In this example three parameters are provided:

- **aMsvSession** – a reference to the previously created messaging session. Note that the session must be created before a CMsvEntry can be instantiated. The CMsvEntry object will use this session to communicate with the message server.
- **aMsvId** – the ID of the entry in the message store that this CMsvEntry will initially be pointing at, in this case it is the local inbox.
- **aOrdering** – the ordering preference for child entries. A CMsvEntry object is often used for enumerating child entries, e.g., for listing all of the messages in a particular folder. The ordering parameter is used to determine the order in which child entries will appear in this list. The summary screen application is only interested in the most recent messages, hence the EMsvSortByDateReverse ordering is used.

In the example code the CMsvEntry object is created and then passed to the summary generator objects. For performance and RAM usage reasons it is desirable not to repeatedly create and delete CMsvEntry objects. Therefore, a reference to the original CMsvEntry is used in this case.


```

class CMessageSummaryGenerator : public CBase
{
public:
    CMessageSummaryGenerator(TMsvId aInboxEntry,
                            TUid aMessageType,
                            CMsvEntry& aMessagingEntry);
    ...
private:
    CMsvEntry& iMessagingEntry;
    TMsvId iInboxEntry;
    TUid iMessageType;
};

```

Example 8.5 The CMessageSummaryGenerator class

Two instances of the class in example 8.5 are created, one for SMS and one for email – each uses a reference to the same CMsvEntry class. The same code is used to generate the summaries for both message types. The aMessageType and aInboxEntry parameters are used to determine which messages are searched for. For SMS messages the message type is set to SMS and the inbox is set to the local inbox; for email the message type is set to POP3 and the inbox is set to the POP3 service entry. See Figure 8.4 for more details on where the different received message types are stored.

The aInbox parameter is used to target the search for new messages at the appropriate folder. Targeting the search at specific folders as opposed to recursively searching the entire message store has two advantages:

1. Any messages in the outbox, draft or any other local folders are ignored.
2. Needlessly searching through a large number of folders and entries can be very time-consuming. Targeting the search at the appropriate folder is much more efficient.

Example 8.6 shows how messages of a particular type can be enumerated:

```

void CMessageSummaryGenerator::StartL(TRequestStatus& aStatus)
{
    aStatus = KRequestPending;

    iMessageSummaries.Reset();

    // Set the CMsvEntry to point to the appropriate inbox, this will be the
    // global inbox for SMS or the remote POP3 inbox for email.
    iMessagingEntry.SetEntryL(iInboxEntry);

    // Get a list of the messages in the selected inbox of the type we are
    // looking for (e.g. SMS or email)
}

```

```
CMsvEntrySelection* filteredMessageIds =
    iMessagingEntry.ChildrenWithMtmL(iMessageType);
CleanupStack::PushL(filteredMessageIds);
... // function continues in Example 8.7 R
```

Example 8.6 Enumerating all messages of a particular type in an inbox

`SetEntry()` is used to set the `CMsvEntry` to point at the appropriate inbox. For SMS this will be the local inbox, for email it will be the POP3 inbox.

Once the correct inbox is selected, `ChildrenWithMtmL()` is used to get the list of child entries of the appropriate message type (SMS or email). It is possible to get the list of all child entries by calling `ChildrenL()`. If the local inbox was selected then `ChildrenL()` would return all of the SMS, MMS and OBEX messages that had been received.

TMsvEntry

The `TMsvEntry` class encapsulates the generic information regarding an entry, for example the date, the subject, etc. It is possible to get the `TMsvEntry` data for a particular entry by using one of several different functions:

- `CMsvSession::GetEntry()` – get the `TMsvEntry` data for any entry from the session.
- `CMsvEntry::Entry()` – get the `TMsvEntry` data for the currently selected message.
- `CMsvEntry::ChildDataL()` – get the `TMsvEntry` data for any child of the currently selected message.

The summary screen application uses `ChildDataL()`, as can be seen in Example 8.7.

```
void CMessageSummaryGenerator::StartL(TRequestStatus& aStatus)
{
    ... // continuing from Example 8.6
    // Calculate the number of messages to summarise
    TInt numberForSummary = filteredMessageIds->Count();
    if (numberForSummary > KMaxSummaryMessages)
        numberForSummary = KMaxSummaryMessages;

    TInt index;
    TMsvEntry tempDataEntry;
    for (index = 0; index < numberForSummary; index++)
        // Generate the message summaries
        {
            // Get the TMsvEntry data for the message at the current index.
```

```
tempDataEntry =
    iMessagingEntry.ChildDataL((*filteredMessageIds)[index]);

// Copy the message details from TMsVEntry to the message summary
// structure (TMessageSummary)
TMessageSummary summary;
summary.iFrom = tempDataEntry.iDetails.Left(KMaxSummaryStringSize);
summary.iSummaryText =
    tempDataEntry.iDescription.Left(KMaxSummaryStringSize);
iMessageSummaries.AppendL(summary);
}

TRequestStatus* status = &aStatus;
User::RequestComplete(status, KErrNone);

CleanupStack::Pop(filteredMessageIds);
}
```

Example 8.7 Retrieving summaries for messages

ChildDataL() is used as opposed to EntryL() because the latter would have required the CMsvEntry to be set to each message entry in turn. As mentioned previously, needlessly changing the context of CMsvEntry can have serious performance implications for the application. This problem is avoided by using ChildDataL() which can extract the TMsVEntry data for the child message entries without the need to repeatedly call CMsvEntry::SetEntry().

In Example 8.7, the summaries for several messages are generated synchronously in one function. This is not a problem in this example, as KMaxSummaryMessages is set to three, so only three messages are checked and the CMsvEntry is not moved. However, if the CMsvEntry::SetEntry() was being called multiple times, or if the entire inbox was being enumerated, then it would be desirable to derive from CActive and break the operation up into several steps, repeatedly setting the object active and then completing it immediately. One step should then be performed in the RunL() each time the object completes.

Message-type specific data

In addition to the generic information encapsulated by the TMsVEntry each message entry also has some message-type specific data, for example email messages contain MIME header information. This message-type specific data is stored and retrieved using a CMsvStore object. The CMsvStore object for a particular message entry is retrieved by calling CMsvEntry::ReadStoreL(). CMsvStore provides similar APIs to the Symbian OS generic store and stream architecture.

A client MTM will usually provide a utility class that can be used to extract the message-type specific data from the CMsvStore, for example CImHeader can be used to extract the MIME headers from an email message.

Some client MTMs also provide additional APIs that can be used to manage attachments without accessing the store directly, for example, the email subsystem provides `CImEmailMessage` that can be used to retrieve the attachment data from a received email message.

Note that retrieving message-type specific data is a much slower operation than retrieving the `TMsvEntry` data. It is advisable to consider the following guidelines in order to achieve good performance and responsiveness:

- Try to group all operations on a particular message in one place. Do not extract one header from each message in turn, and then do the same for a different header. Instead, extract all required headers from one message then move to the next one.
- Implement iterative code as an active object where each message or attachment is processed in a separate call to the `RunL` function. This is necessary to ensure that the application remains responsive even when enumerating a large number of messages or attachments.

MMsvSessionObserver

Previous code examples have shown how the `MMsvSessionObserver::HandleSessionEventL()` interface can be used to determine when the messaging session has completed its initialization. After the session has been created, then this function is also called to notify the observer of any subsequent changes to the messaging store. Examples include entry creation or deletion, and changes to existing entries.

Example 8.8 shows how `HandleSessionEventL()` can be used to refresh the summary screen application when a new message has been received:

```
void CMessageSummaryEngine::HandleSessionEventL(TMsvSessionEvent aEvent,
                                                TAny *aArg1, TAny *aArg2, TAny * /* aArg3 */)
// This function is called by the message server for every message server
// event, e.g. when a new message is received or created. */
{
    switch (aEvent)
    {
        ...
        case MMsvSessionObserver::EMsvEntriesChanged:
            // Get the list of the changed messages. Note
            // that there may be more than one.
            CMsvEntrySelection* newMessages = static_cast<CMsvEntrySelection*>
                (aArg1);

            // Get the ID of the parent entry
            TMsvId* parentMsvId = static_cast<TMsvId*> (aArg2);

            // Set the CMsvEntry to point to the parent entry
            iMessagingEntry->SetEntryL(*parentMsvId);
```

```

// Add each new message to the appropriate message summary
TInt index = newMessages->Count() - 1;
while (index >= 0)
{
    // Get the entry details for each new message
    TMsVEntry tempMessageEntry =
        iMessagingEntry->ChildDataL(newMessages->At(index));

    if (tempMessageEntry.Complete()) // Is the message
    // complete? Do not add incomplete entries.
    {
        if (iSmsInboxSummary->MessageOfCorrectType(tempMessageEntry))
        {
            iRefreshSms = ETrue;
        }
        else if ((iEmailInboxSummary != NULL) && (iEmailInboxSummary->
            MessageOfCorrectType(tempMessageEntry)))
        {
            // If we have a new email message in the email inbox then
            // update the email summaries
            iRefreshEmail = ETrue;
        }
    }

    index--;
}

if ((iRefreshSms) || (iRefreshEmail))
    // Refresh the SMS and / or email list if there have been any
    // changes to SMS and / or email messages
    {
        iState = ERefreshing;

        // If this object is not active then update the next summary
        // If this object is active then the appropriate summary list
        // will be refreshed when it reaches the RunL.
        if (!IsActive())
            UpdateNextSummary();
    }
break;

```

Example 8.8 Monitoring the message store for updates

It is possible that a message server plug-in (MTM) may create an incomplete entry in the inbox while the incoming message is being processed. It is not desirable for the summary screen application to display this incomplete message. In order to get around this problem the above code checks for the `MMsvSessionObserver::EMsvEntriesCreated`. `CMsvEntry::Complete()` is then called to check that the changed message entry has been flagged as complete. The summary screen is only refreshed if this flag is set.

Note that this code will not currently update the inbox if a message is deleted. In order to add this functionality, an additional case statement is required to handle the `EMsvEntriesDeleted` notification.

8.4 Messaging Application Design and Implementation

When designing a messaging-aware application it is essential to consider how it will access the message store. If messaging APIs are used inappropriately then it is possible to seriously degrade the performance of the application and potentially the performance of the device. Some things to consider when designing a messaging-aware application are listed below.

8.4.1 Asynchronous Behaviour

It is usually advisable to make messaging operations as asynchronous as possible. This is desirable because many messaging operations can potentially be long running, for example, enumerating all messages in the inbox, which may number in the hundreds. If potentially large numbers of messages are to be processed then an active object should be used, processing only a small number of messages – about 10 would be a recommended value – in each `RunL()` call. This will help to ensure that the application remains responsive to user input and that it can be cancelled at any time.

The summary screen application uses an active object to refresh the SMS and email summaries in two steps. This also allows the single `CMsvEntry` to be shared between both objects, as they perform their operations sequentially.

```
class CMessageSummaryEngine : public CActive, public MMsVSessionObserver
{
public:
    ...
    // Start the summary operation
    void StartL();

    // CActive functions
    void DoCancel();
    void RunL();
    ...
};
```

Example 8.9 Class declaration for `CMessageSummaryEngine`

The `RunL()` is called once for each message type. This means that the operation can be cancelled without waiting for all of the summaries to be updated. It also allows other active objects to run, for example any active objects that may be responsible for updating the UI. This would be even more important if more message types were enumerated in addition to email and SMS, or when there are a significant number of emails or SMSs in a folder.

```
void CMessageSummaryEngine::RunL()
{
    // One of the message summary generators has completed.

    if (!UpdateNextSummary())
    {
        // If UpdateNextSummary returns EFalse then there are no more message
        // types to check. So refresh the UI.
        iObserver.Refresh();
    }
}
```

Example 8.10 Updating the summary screen in stages to prevent the UI becoming unresponsive

In addition to using active objects to break up large iterative tasks into smaller steps, it is usually desirable to use the asynchronous version of messaging APIs in cases where both synchronous and asynchronous versions are available.

8.4.2 Applying Changes to the Message Store

Two things happen every time a message store entry is changed:

1. The observer of every messaging session in the system is notified of the change.
2. There is a significant volume of file system access.

When applying changes to the message store it is recommended that they are submitted as one operation. For example, it is not desirable to repeatedly call `CMsvEntry::ChangeL()` to modify individual `TMsvEntry` flags. A better approach is to collate all of the changes required to the `TMsvEntry` and then to apply them all at once.

8.4.3 Using `CMsvSession` and `CMsvEntry`

Some key recommendations regarding the messaging classes are listed below. It is worth considering these general rules when designing a messaging-related application.

- Only create one `CMsvSession` object per application. If you need more observers then use the `CMsvSession::AddObserverL()` function.
- Do not add unnecessary observers. Note how a single observer is used in the summary screen application. This observer then delegates to other classes depending on the contents of each notification.

- Do not move a `CMsvEntry` object to point to different entries unnecessarily.
- Do not leave a `CMsvEntry` pointing at an entry that another application may want to delete. For example do not leave a `CMsvEntry` pointing at an entry in the inbox because this will prevent the user from deleting this message.
- Do not leave a `CMsvEntry` pointing at a folder which contains lots of child entries (e.g., the inbox) as this uses lots of RAM.

8.5 Receiving Application-specific SMS Messages

The messaging APIs usually provide the most convenient access to messaging-related functionality. However, there are some cases where these APIs are not appropriate. The most common example of this is when it is necessary to intercept a message before it is written to the message store, as the addition of messages to the inbox typically triggers a user notification.

The example summary screen application implements a weather report function that is activated when a specially formatted SMS is received. One approach would be to use a messaging observer to wait for a new SMS message to appear in the inbox. When the new SMS is received, `HandleSessionEventL()` would be called. At this point some code could check to see if it is a weather report message. If so, then the content of the message would be passed to the weather report parser and the message would then be deleted. Unfortunately there are some disadvantages to this approach:

1. The binary message, which is not intended for the user in raw form, briefly appears in the inbox before disappearing again.
2. All observers in the system are notified about the binary SMS and its subsequent deletion. This may result in the user being alerted to the arrival of a new SMS.

A better approach for implementing this functionality is to intercept the SMS message before it reaches the message server. This can be achieved by using the SMS socket API.

8.5.1 SMS Socket API

In the summary screen application, the `CWeatherReportWatcher` class waits for weather report messages. It does this by opening a socket

to the SMS stack² and registering for all SMS messages starting with 'Weather:'.

The 'Weather report' SMS encoding is very simple – it consists of the prefix 'Weather:' followed by a single digit, either '1', '2', or '3'. These correspond to 'sunny', 'cloudy' or 'rainy' – we don't get much interesting weather in the UK!

Example 8.11 shows the declaration of `CWeatherReportWatcher` that is later used to intercept the weather report messages.

```
class CWeatherReportWatcher : public CActive
{
    ...
private:
    RSocketServ iSocketServer;
    RSocket iSmsSocket;

    // ...

    enum TWeatherWatcherState
    {
        EWaitingForWeatherReport,
        EAcknowledgingWeatherReport
    };

    TWeatherWatcherState iState;
};
```

Example 8.11 Class declaration for `CWeatherReportWatcher`

Example 8.11 shows how the `CWeatherReportWatcher` has two states.

1. Waiting for a weather report SMS message.
2. Acknowledging the received SMS message.

The weather report watcher is implemented as an active object because both the waiting and the acknowledging operations are potentially long running and therefore should be performed asynchronously. The active object allows these operations to run without blocking the entire thread. Note that if the thread was blocked then the message summaries would not be updated with any newly received messages.

Example 8.12 shows how the SMS socket is opened and how it registers an interest in SMS messages with the prefix "Weather:" by using `RSocket::Bind()`.

```
_LIT8(KWeatherReportPrefixString, "Weather:");
...
```

² For generic information on socket usage, see Chapter 3.

```

void CWeatherReportWatcher::ConstructL()
{
    CActiveScheduler::Add(this);

    // Connect to sockets server
    // SMS messages are intercepted via sockets
    User::LeaveIfError(iSocketServer.Connect());

    // Open SMS socket
    User::LeaveIfError(iSmsSocket.Open(iSocketServer, KSMSAddrFamily,
        KSockDatagram, KSMSDatagramProtocol));

    // Set SMS prefix - only intercept SMS messages starting with a
    // particular string
    TSmsAddr smsAddress;
    smsAddress.SetSmsAddrFamily(ESmsAddrMatchText);
    smsAddress.SetTextMatch(KWeatherReportPrefixString);
    iSmsSocket.Bind(smsAddress);

    WaitForWeatherReportL();
}

```

Example 8.12 Registering to receive SMS messages with a specific prefix

`WaitForWeatherReport()` implements the code that actually waits for a weather report message to be received. This is shown in Example 8.13. Note how `RSocket::Ioctl()` is used to asynchronously wait for an SMS that matches the attributes that have already been set on the socket.

```

void CWeatherReportWatcher::WaitForWeatherReportL()
{
    // Wait for an appropriate SMS message
    // the RunL will be called when an appropriate SMS message is received

    // Note that the smsAddress has already been set in ConstructL so we
    // will only intercept messages starting with 'Weather:'

    iSbuf()=KSockSelectRead;

    iSmsSocket.Ioctl(KIOctlSelect, iStatus, &iSbuf, KSOLSocket);
    iState = EWaitingForWeatherReport;

    SetActive();
}

```

Example 8.13 Waiting for a message with the registered prefix to arrive

When a weather report message has been received `RunL()` is called. `RunL()` is split into two parts, the first to handle a newly received weather report and the second to handle the completion of the acknowledgement.

```

void CWeatherReportWatcher::RunL()
{
    if (iState == EWaitingForWeatherReport)
    {
        CSmsBuffer* smsBuffer=CSmsBuffer::NewL();
        CleanupStack::PushL(smsBuffer);

        CSmsMessage* smsMessage = CSmsMessage::NewL(iFs, ESmsDeliver,
            smsBuffer);

        // smsMessage has taken ownership of smsBuffer so remove it from the
        // cleanup stack.
        CleanupStack::Pop(smsBuffer);

        CleanupStack::PushL(smsMessage);

        RSmsSocketReadStream readstream(iSmsSocket);

        // This function may leave
        readstream >> *smsMessage;

        // Extract the text from the SMS buffer
        TBuf<KMaxWeatherReportLength> weatherReportBuf;
        TInt bufferLength = smsBuffer->Length();
        if (bufferLength > KMaxWeatherReportLength)
        {
            bufferLength = KMaxWeatherReportLength;
        }
        smsBuffer->Extract(weatherReportBuf, 0, bufferLength);
    }
}

```

Example 8.14 Handling an SMS upon arrival

Example 8.14 shows how the contents of the SMS message are extracted from the SMS socket and how CSmsBuffer and CSmsMessage are used. Once the weather report has been extracted into a descriptor then it is processed by the code shown in Example 8.15:

```

MWeatherReportObserver::TWeatherReport weatherReport =
    MWeatherReportObserver::ENone;

// Process the message
if (weatherReportBuf.Length() >= KMaxWeatherReportLength)
{
    // Get the last character. The last character represents the
    // weather report.
    TUint16 lastCharacter = weatherReportBuf[KMaxWeatherReportLength -
        1];

    // Process the message
    switch (lastCharacter)
    {
        case '1':
            weatherReport = MWeatherReportObserver::ESunny;
            break;
        case '2':
            weatherReport = MWeatherReportObserver::ECloudy;
            break;
    }
}

```

```

case '3':
    weatherReport = MWeatherReportObserver::ERainy;
    break;

    // No default. Leave weather report as 'None'.
}
}

// Update the UI with the new weather report
iWeatherReportObserver.NewWeatherReport(weatherReport);

CleanupStack::PopAndDestroy(smsMessage);

```

Example 8.15 Processing the received SMS

Messages with the contents “Weather:1” cause the UI to be updated with an ESunny weather report. Messages with contents “Weather:2” cause the UI to be updated with an ECloudy report, etc.

Once the incoming message has been processed then the application must acknowledge it. This lets the SMS stack know that the application has successfully dealt with the SMS so it can be deleted from the reassembly store.

If the current state of the system prevents the handling of this message, for example, due to low memory then the acknowledgement can be omitted so that the SMS stack will attempt to deliver the message to the application again and it can be processed later when the appropriate resources may be available.

Example 8.16 shows how the weather report is acknowledged.

```

// Acknowledge successful processing of the SMS message
TPkgBuf<TUint> sbuf;
iSmsSocket.Ioctl(KIoctlReadMessageSucceeded, iStatus, &sbuf,
                KSolSmsProv);

// Wait for the acknowledgement to be sent, go active.
iState = EAcknowledgingWeatherReport;
SetActive();
}

```

Example 8.16 Acknowledging receipt of a received SMS

Finally, CWeatherReportWatcher waits for the acknowledgement to complete. Once this is done RunL() is called again, this time the acknowledgement branch is executed where it then waits for a new weather report message.

```

void CWeatherReportWatcher::RunL()
{
    ...
    else if (iState == EAcknowledgingWeatherReport)

```

```
{  
    // The acknowledgement has now been sent so wait for another weather  
    // report.  
    WaitForWeatherReportL();  
}  
}
```

Example 8.17 Waiting for the acknowledgement process to complete

8.6 Summary

In this chapter, we have learnt how to:

- define the structure of the message store
- access messages in the message store
- filter messages, for example by message type
- access the generic parts of each message, for example, sender and subject information
- access the message-type specific parts of messages, such as, email attachments
- monitor the message store for changes
- receive an application-specific SMS before it reaches the message store.

9

Sending Messages

This chapter builds on Chapter 8 to provide a more complete guide to the messaging architecture on Symbian OS.

This chapter details the high-level SendAs messaging API in Symbian OS and also the UI platform-specific Send dialogs. The chapter concludes with an example of how to extend the messaging architecture by implementing a Flickr MTM, building upon the HTTP example code provided in Chapter 11 to upload an image to Flickr.

The chapter is useful for developers wishing to implement a Send menu in their application and to those wishing to extend the Send menu. It provides a starting point to implementing a fully fledged set of MTMs.

There are four main use cases which might have you looking at the messaging APIs:

1. Implementing services such as SMS-based services/games, or inbox cleaners.
2. Using the SendAs API to transfer content between devices.
3. Extending the options on the Send menu.
4. Implementing a new message transport such as a push email MTM.

We saw how to implement (1) in Chapter 8. The first part of this chapter covers (2), and the second part covers (3) fully, as well as providing a basis for (4).

Be aware that implementing new message transports requires your code to be trusted with capabilities from the extended capability set, including `ReadDeviceData`, `WriteDeviceData`, `ProtServ`

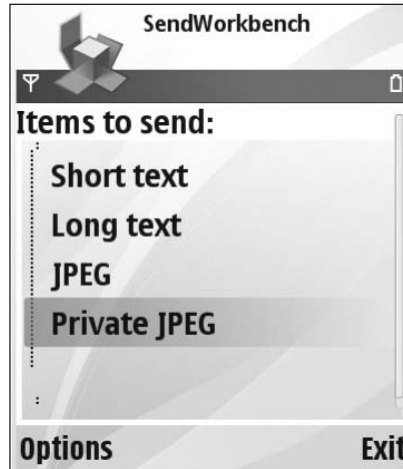


Figure 9.1 The SendWorkBench application on S60

and NetworkControl. It is advisable to investigate the signing requirements for such code before starting implementation.

9.1 Examples Provided in this Chapter

There are two main parts to the example code for this chapter.

The first is a SendWorkBench application (with S60 and UIQ versions), which demonstrates the use of SendAs to create SMSs and emails, and SendUI to offer the user a choice of how to send one message with text and one with attachments (see Figure 9.1).

The second is an implementation of a set of MTMs, which add a ‘Send to Flickr’ item to the Send menu of S60 and UIQ.

9.2 SendAs Overview

9.2.1 SendAs Messaging

SendAs is an API which presents a simple abstraction over the various messaging technologies found in mobile devices.

SendAs extends the messaging architecture and adds a new layer on top, enabling the discovery and use of protocols for sending data from the device. An application using SendAs describes the type and size of data it would like to send, and can discover message types that are able to accommodate the payload.

SendAs can also launch a message editor to allow the user to modify the message before sending, as in the case of following a ‘mailto:’ link in a web browser – the browser creates an empty email with the recipient field filled in.

Different methods of sending may have varying costs to the user. An SMS is typically charged per message, while sending an email over WLAN often adds no extra cost – therefore an application usually offers the user a choice of compatible bearers.

SendAs can be used directly to create this list, but UI platforms provide wrappers around this functionality to allow easier integration into a UI application.

9.2.2 The Send Dialog

UIQ and S60 provide UI-level APIs for adding a Send item to an application’s menu, layered on top of the SendAs API. The term ‘SendUI’ is often used to refer to the DLL which provides these services, but for the purpose of this chapter the term Send menu refers to the menu or dialog provided by the respective UI (see Figure 9.2).

On S60 the Send dialog API is provided by the `CSendUi` class and `sendui.lib`; on UIQ it is provided by `CQikSendAsDialog`, `CQikSendAsLogic` and `qikutils.lib`.

A Send dialog offers the user a choice of sending methods compatible with the data being sent. The choice given to the user may further be filtered if no accounts exist for a particular MTM, for example, there is



Figure 9.2 Send menus on S60 3rd edition and UIQ3

no point allowing the user to ‘send via email’ if no email accounts have been defined.

Both SendAs and the various Send dialog APIs take advantage of the fact that all bearers can handle data as either text or as an attachment to provide a uniform API for querying capabilities and creating messages.

9.2.3 When to use SendAs

SendAs and the Send dialogs have similar roles in that both UIQ and S60 classes wrap up message creation in a similar way. The Send dialogs require an application environment (CEikonEnv), and are not suited to a background service, whereas any thread or process can use RSendAs.

- Use the Send dialog when your application has data to send, but you want to let the user choose which transport to use.
- Use RSendAs when you know the transport and contact address, for example, when sending an SMS to a fixed number.

9.2.4 SendAs Overview

RSendAs and RSendAsMessage are the main classes which make up the Symbian ‘SendAs’ API. They provide the following services:

- list all MTMs installed or those that have specific capabilities
- list services/accounts available for each MTM
- create and send a message with confirmation from the user
- create and send a message without confirmation from the user (if the application does not have the capabilities required to send a message using a given MTM then the user will be prompted as to whether to allow the message to be sent)
- create and launch the appropriate editor, delegating the modification and sending of the message to the editor
- create and save messages to the drafts folder for opening from the messaging application.

9.2.5 Messaging on a Real Device

Different devices are likely to have different sets of MTMs installed – the choice is down to the device manufacturer. Equally, the emulator supplied in the various SDKs may have a variable set of MTMs installed. Therefore

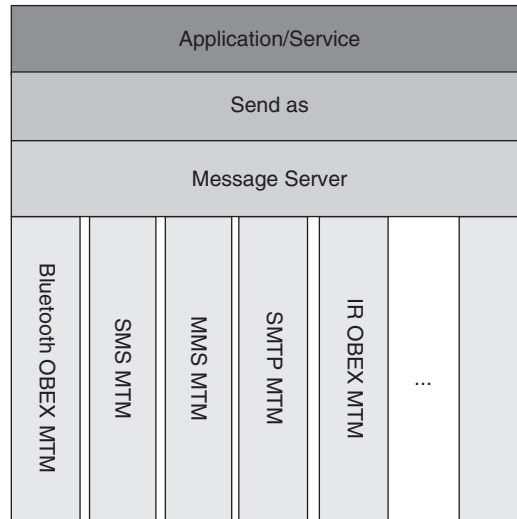


Figure 9.3 High-level diagram of a messaging subsystem on a typical phone

it's not always possible to guarantee that a particular bearer is available on a device. An obvious example would be the IR OBEX MTM. If a device has no IrDA transceiver then the IR OBEX MTM will not be built in to the device. Unless you are targeting a specific device, then it's best not to make assumptions about installed MTMs.

Figure 9.3 shows a typical array of MTMs installed on a device.

It's worth noting that the MMS MTM has different implementations and APIs on UIQ and S60. Using SendAs will shield you from these differences.

9.2.6 Basic Capabilities

Each MTM reports whether it supports body text and attachment attributes. (see the table below).

Bearer	Supports body text	Supports attachment
Bluetooth	×	✓
Infrared	×	✓
SMS	✓	×
MMS	×	✓
Email	✓	✓

9.3 Services/Accounts

Each MTM may have zero or more services associated with it. The purpose of a service is to store settings or account details for an MTM. An email account is defined by two services – one for incoming messages (POP3/IMAP) and one for outgoing messages (SMTP). If no account is supplied to SendAs, or Send dialog, then the default account is used. The concept of a default account is consistent throughout all Symbian OS-based devices, so unless there is a specific reason then SendAs clients should always stick to the default.

9.4 Technical Description

Figure 9.4 shows how SendAs fits into the overall messaging architecture. There are two process boundaries in place in order to allow policing of capabilities at both the message server interface and for clients of SendAs.

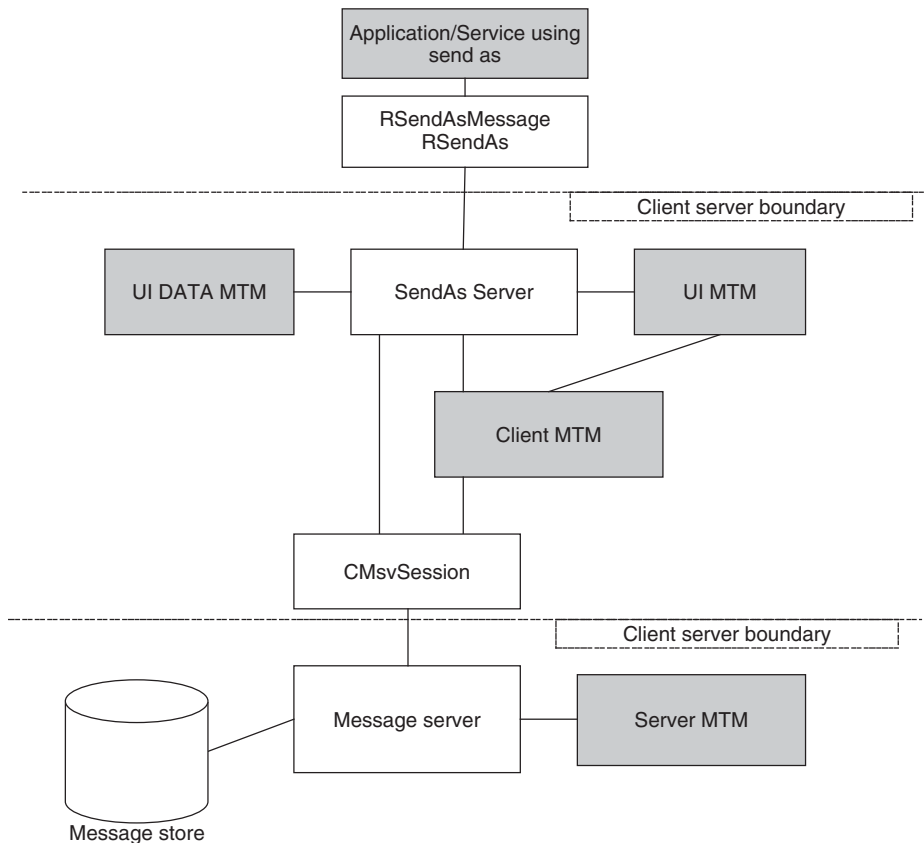


Figure 9.4 Integration of the SendAs service with the messaging architecture

The application/SendAs server process boundary means that whilst the SendAs server must have the platsec capabilities to write messages to the message store, SendAs clients do not.

If a SendAs client without the appropriate capabilities attempts to send a message, then SendAs will first prompt the user to confirm the send.

A client which wishes to silently send messages from a background service must have all the capabilities that are required by both the message server and the MTM being used.

S60 3rd edition and S60 3rd edition Feature Pack 1 use a slightly modified version of this architecture without the interim SendAs server. This means that the client-side MTMs need to be signed with enough capabilities to be loaded into the client process, rather than being loaded into the SendAs server process. More details can be found in the S60 3rd edition SDK documentation, including a list of the capabilities required by the client-side MTMs.

9.4.1 Sending a One-shot SMS

To give a flavour of the practical use of SendAs, the following example code fragment sends an SMS to a fixed, fictional, phone number.

```
//Link against sendas2.lib
#include <smut.h> // for KUidMsgTypeSMS
#include <sendas.h>
#include <rsendasmessage.h>

void SendAnSMSL()
{
    _LIT(KSMSRecipient, "+447700900007");
    _LIT(KSMSBodyText, "Some body text to fill up space");

    RSendAs sendAs;
    User::LeaveIfError(sendAs.Connect());
    CleanupClosePushL(sendAs);

    RSendAsMessage sendAsMessage;
    sendAsMessage.CreateL(sendAs, KUidMsgTypeSMS);
    CleanupClosePushL(sendAsMessage);

    // prepare the message
    sendAsMessage.AddRecipientL(KSMSRecipient),
        RSendAsMessage::ESendAsRecipientTo);
    sendAsMessage.SetBodyTextL(KSMSBodyText);
    // send the message
    sendAsMessage.SendMessageAndCloseL();
    CleanupStack::PopAndDestroy(2, &sendAs);
}
```

Creating the message and adding the recipient is pretty straightforward: `ESendAsRecipientTo` is used to denote that the recipient should be added to the To: list, which for SMS is the only list.

If we were sending via email then `ESendAsRecipientCc` or `ESendAsRecipientBcc` could be specified for carbon copy and blind carbon copy recipients, respectively.

`RSendAsMessage::SendMessageAndCloseL()` starts sending the message and closes the handle. It does not wait for the message to be sent. `SendMessageConfirmedAndCloseL()` does the same, but gets user confirmation for sending even if the process has the correct capabilities.

Finally, `LaunchEditorAndCloseL()` may be used to launch the appropriate message editor. In the above example the user would be presented with a message containing a recipient and body text which can be edited. A `SendAs` client who launches an editor delegates ownership of the message to the editor. It makes little sense for the client to keep track of the message after this point, as the user can edit, save to drafts or delete the message at any time.

9.4.2 Asynchronous Sending

A `SendAs` client which wishes to know if a message send succeeded must use the asynchronous methods `RSendAsMessage::SendMessage()` and `RSendAsMessage::SendMessageConfirmed()`. Both take a `TRequestStatus` that will be completed with an error code in the event of failure, or completed with `KErrNone` if sending was successful.

9.4.3 Creating Attachments

Attachments and data caging

Pre Symbian OS v9.0, the message store allowed each message entry to have a folder in the mail store and write or copy attachment files at will. These attachments were visible to all and could be opened by navigating through the mail folder using a third-party file browser.

The introduction of data caging means that the message server now owns and polices access to a private message store – applications can no longer get direct access and must use message server APIs to be able to access or add attachments.

Only the message server may access the private directory structure that makes up the message store, but it's useful to know the location (which may change in the future) for debugging purposes when using the emulator.

```
\c\private\1000484b\Mail2\
```

Several mechanisms are provided to facilitate reading and writing of attachment data to the message store. For RSendAs clients, there are three ways of attaching files to a message. It's up to the SendAs client to decide which is most appropriate for the data it is presenting.

The SendWorkbench example project demonstrates the different methods of creating attachments using the SendAs API. SendWorkbench exports an image file to a public directory and a private directory.

```
_LIT(KPrivateAttachmentFile, "C:\\private\\20009979\\sendasexample.jpg");
_LIT(KPublicAttachmentFile, "c:\\data\\images\\sendasexample.jpg");
```

Send attachment by copy

This is appropriate when an application stores reasonably-sized (less than a few 100 kB) data files, possibly in its private directory, and wants to send the file as is by letting RSendAs create a copy of the file in the message store.

This method requires enough free disk space for an extra copy of the file.

```
void CSender::SendAttachmentL()
{
    RSendAs sendAs;
    User::LeaveIfError(sendAs.Connect());
    CleanupClosePushL(sendAs);

    RSendAsMessage sendAsMessage;
    sendAsMessage.CreateL(sendAs, KUidFlickrMtm);
    CleanupClosePushL(sendAsMessage);

    TRequestStatus status;
    sendAsMessage.AddAttachment(KPublicAttachmentFile, status);
    User::WaitForRequest(status);

    // send the message
    sendAsMessage.SendMessageAndCloseL();
    sendAs.Close();
    CleanupStack::PopAndDestroy(2, &sendAs);
}
```

An attachment may also be copied by file handle using the overload of AddAttachment() which takes an RFile parameter – if the file to be copied resides in an application's private directory it is necessary to use this version of AddAttachment().

Send as linked attachment

This is ideal for large user data items stored on public areas of the disk. A good example is large JPG images, which in this example we'll say are stored in c:\data\images.

Rather than waste time and disk space by copying the file to the message, only the full filename of the file is stored with the message.

If the source file is deleted, removed or renamed then the link will be broken. It's up to the MTM what it does if the attachment is missing at the time of sending.

```
_LIT(KPublicAttachmentFileNameAndPath, "c:\\data\\images\\sendatest.jpg");
sendAsMessage.AddLinkedAttachment(KPublicAttachmentFileNameAndPath,
    iStatus);
```

Building an attachment from another source

Often an application needs to externalize data to a particular form before attaching it; for instance, the contacts engine must externalize contacts to a VCF file in order to send them over infrared or Bluetooth.

A file can be built by writing data to a file handle provided by the message server. This avoids writing to a temporary file and trying to attach it.

```
_LIT(KPublicAttachmentFileName, "sendatest.jpg");
_LIT(KPublicAttachmentFileNameAndPath,
    "c:\\data\\images\\sendatest.jpg");

RFile attachmentFile;
sendAsMessage.CreateAttachmentL(KPublicAttachmentFileNameAndPath,
    attachmentFile);

RFs fs;
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);

RFile input;
User::LeaveIfError(input.Open(fs, KSendAsTestAttachmentNameAndPath,
    EFileRead| EFileShareReadersOnly));
CleanupClosePushL(input);

TInt fileSize=0;
User::LeaveIfError(input.Size(fileSize));
HBufC8* fileBuf = HBufC8::NewMaxLC(fileSize);
TPtr8 ptr(fileBuf->Des());
User::LeaveIfError(input.Read(0,ptr,fileSize));

User::LeaveIfError(attachmentFile.Write(ptr));
CleanupStack::PopAndDestroy(3, &fs);
handle.Close();
```

9.5 Using the UI Platform Send Dialogs

This section briefly explains how Send dialog classes are used to send messages and attachments by allowing the user to specify the bearer.

A common task for an application developer is to add a Send item to the application menu. The SendWorkBench example shows how to use SendUI to present the user with a choice.

9.5.1 Send dialog on S60 3rd edition

On S60 3rd edition, the SendUI functionality is available to any application that wishes to export data. For instance, Figure 9.5 shows the results of choosing the Send from the standard contacts application.

The contact card can be sent as either a vCard attachment or as a 'smart' text message. The Send menu offers a choice of bearers which reflect this. In Figure 9.5 email is not listed as a bearer as no accounts have been created on this particular device.

*Creating a **SendUI** instance*

Create a CSendUi object as part of the application UI constructor. An instance of CSendUi is kept for the lifetime of the app UI since its services are called repeatedly whenever the menu is displayed.

```
iSendUI = CSendUi::NewL();
```

Adding a send menu item

It is possible to hardcode a Send item in the menu, but using SendUi ensures that the menu text is localized correctly.

In the fragment of code below, the new Send menu item sits at the top of the menu pane (Item index 0)

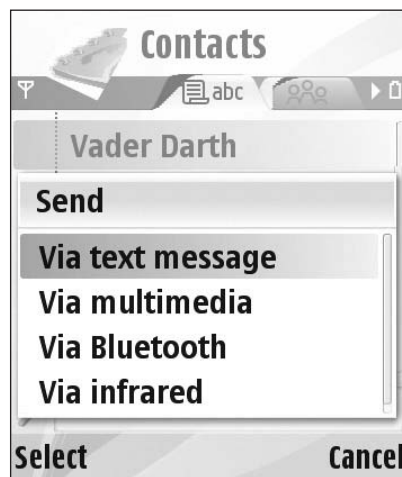


Figure 9.5 Sending a contact using the Send dialog on S60


```
void CSendWorkbenchAppUi::DynInitMenuPaneL(TInt aResourceId, CEikMenuPane
    *aMenuPane)
{
    const KSendMenuItemIndex = 0;
    switch (aResourceId)
    {
        case R_MENU:
        {
            iSendUI->AddSendMenuItemL(*aMenuPane, KSendMenuItemIndex,
                                     EOwnCommandSend);

            break;
        }
    }
}
```

Creating a message and setting the body text

The following fragment shows how to set up some simple rich text for sending.

```
_LIT(KTextSmall, "This is a small message which should easily fit into a
    single SMS");

CRichText* PrepareSmallTextLC()
{
    CEikonEnv& ee = *CEikonEnv::Static();

    CRichText* rt = CRichText::NewL(ee.SystemParaFormatLayerL(),
        ee.SystemCharFormatLayerL());
    CleanupStack::PushL(rt);
    rt->InsertL(0, KTextSmall);

    return rt;
}
```

CMessageData is a key class in the SendUi API – it is populated with the data and is passed to CSendUi as the payload.

```
CMessageData* data = CMessageData::NewLC();
CRichText* rtSmall = PrepareSmallTextLC();
const TInt docLength = rt->DocumentLength();
sc = TSendingCapabilities(docLength, docLength,
    TSendingCapabilities::ESupportsBodyText);
data->SetBodyTextL(rtSmall);
```

Attaching a file

```
const KMaximumSizeUnknown = 0;
sc = TSendingCapabilities(KMaximumSizeUnknown, KMaximumSizeUnknown,
    TSendingCapabilities::ESupportsAttachments);
data->AppendAttachmentL(KPublicAttachmentFile); <KPublicAttachmentfile
```

Sending the message

```
iSendUI->ShowQueryAndSendL(data, sc);
```

ShowQueryAndSendL() does not take ownership of data, but copies the data to the message store. It's safe to delete data as soon as ShowQueryAndSendL() returns.

Attaching a file from the private directory

A file in the private directory of the application is not accessible to other processes, including the SendAs and messaging servers, and so it must be added by handle.

```
const KMaximumSizeUnknown = 0;
sc = TSendingCapabilities(KMaximumSizeUnknown, KMaximumSizeUnknown,
                        TSendingCapabilities::ESupportsAttachments);

RFs fs;
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);
fs.ShareProtected();

RFile file;
User::LeaveIfError(file.Open(fs, KPrivateAttachmentFile, EFileRead));

CleanupClosePushL(file);

data->AppendAttachmentHandleL(file);
CleanupStack::PopAndDestroy(2, fs);
```

KPrivateAttachmentFile is the fully qualified filename for the attachment to be sent.

9.5.2 Send dialog on UIQ3.0

Adding a send menu item

Unlike S60, a UIQ application defines its Send menu item as a standard menu resource.

SendWorkBenchUIQ contains the following resource:

```
RESOURCE QIK_COMMAND_LIST r_SendWorkbenchUIQ_commands
{
    items =
    {
        QIK_COMMAND
```

```
{
    id = ECommandSendItem;
    type = EQikCommandTypeScreen;
    text = STRING_r_SendWorkbenchUIQ_send_item_cmd;
}

};

/* more items here */
}
```

Creating a message and setting the body text

CQikSendAsLogic is the simplest way of creating message content for use with the Send dialog. It contains methods for associating rich text and attachments with a message.

```
iSendAsLogic = CQikSendAsLogic::NewL(iSmallText, KNullDesC, KSmallText);
```

iSmallText is a CRichText instance whose ownership is transferred to iSendAsLogic. iSmallText will be deleted when the message has been committed to the message store. The second parameter KNullDesC indicates that there are no attachments, and KSmallText is simply the subject of the message.

Attaching a file

```
TUId capability;

iSendAsLogic = CQikSendAsLogic::NewL(NULL, KAttachmentSource,
                                       KAttachmentText);
capability = KUIdMtmQuerySupportAttachments;
```

The (messaging, not platsec) capability requests that only MTMs that support attachments should be presented when the Send dialog is shown (capability is used below).

Attaching a file from the private directory

Adding a file from a private application directory involves creating a file server session, an open file handle and transferring the ownership of the handle to the send as logic.

```
iSendAsLogic = CQikSendAsLogic::NewL();
capability = KUidMtmQuerySupportAttachments;

RFs fs;
User::LeaveIfError(fs.Connect());
CleanupClosePushL(fs);
fs.ShareProtected();
RFile file;

User::LeaveIfError(file.Open(fs, KPrivateAttachmentFile, EFileRead));
CleanupStack::Pop(&fs);
iSendAsLogic->AddAttachmentL(file);
```

KPrivateAttachmentFile is the fully qualified file name of the attachment to be sent.

Invoking the SendAs dialog to send the message

With the iSendAsLogic set up by the application in one of the ways detailed above, the UIQ SendAs dialog has all the information it needs to send the message, aside from which bearer should be used.

The following call takes care of sending the message once it has prompted the user.

```
CQikSendAsDialog::RunDlgLD(iSendAsLogic, capability);
```

The dialog takes ownership of CQikSendAsLogic and so deletes it before returning.

9.6 A Brief Background to MTMs

The messaging architecture was designed as an extensible system with plugin modules known as MTMs. An MTM is a group of DLLs which provide support for a particular message type. The messaging architecture allows client applications to determine at runtime which MTMs are installed on the device and when new MTMs are installed.

9.6.1 History

One of the goals of the original messaging architecture (which dates back to the Psion Series 5) was to allow smooth integration of an after-market IMAP4 solution for the Psion Series 5mx.

A mantra used in the creation of the architecture was that the main messaging application (or message centre) should contain no bearer specific code, dealing only with abstract concepts of messages.

The result is that the MTM framework is a delegation framework which yields ‘chatty’ interactions between a messaging clients and the set of installed MTMs. The client can ask if an MTM supports a particular capability and then take action.

Since the IMAP4 MTM was developed, other technologies have been integrated into the messaging framework: OBEX over IR and OBEX over Bluetooth MTMs, MMS, and email sync MTMs have been successfully integrated into OEM devices.

The MTM architecture is heavily segmented by transport type rather than message type which can cause some misunderstanding for developers attempting to introduce a new message type.

A full set of MTM plugins consists of an end-to-end solution for a message type and transport. This includes an editor to create messages, a class to supply icons and a description for items in the list view of the messaging application, a class to store and manipulate messages, and a class for sending and receiving a message. Most of these elements appear in figure 9.6, along with details of how they interact.

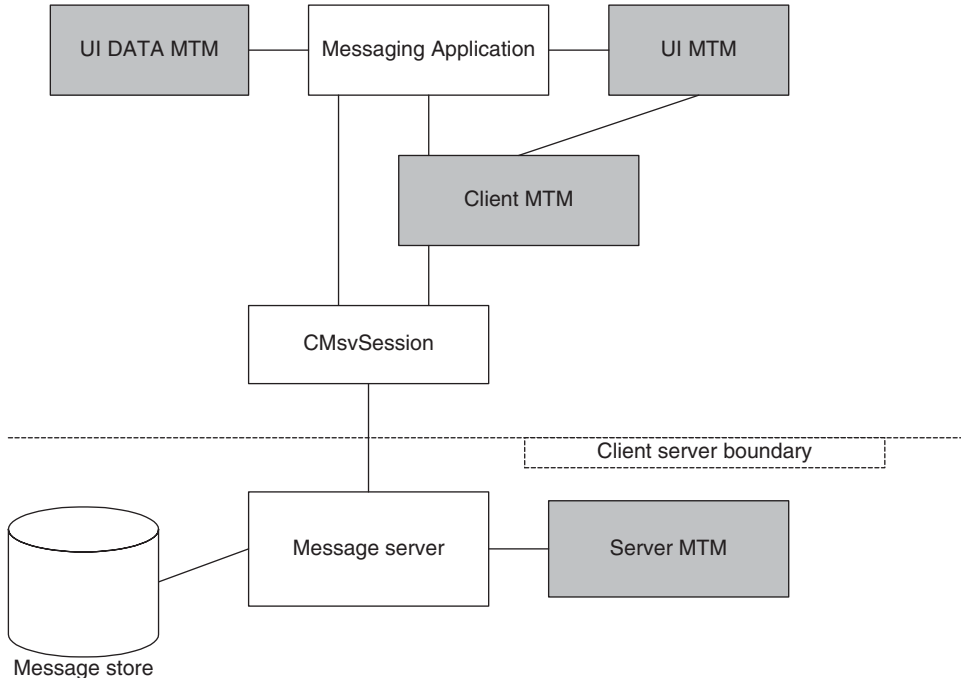


Figure 9.6 High-level overview of the messaging architecture

9.6.2 MTM diagram

Why four DLLs?

In theory, it is not necessary to put each component in a separate DLL. The MTM registration resource file allows the entry point for each MTM to be defined, enabling packaging of one or more MTM components into a single DLL. However, the MTMs are often split into their own DLLs for a variety of reasons:

- The MTM client-side DLL should remain separate from the server side for platsec reasons: a server-side DLL will require one set of capabilities – those of the message server process – whereas a client-side DLL is likely to require a different set of capabilities – in theory those of any application process into which it might be loaded. See the earlier discussion about the SendAs architecture for information about which DLLs are loaded into which processes.
- The client and server MTMs are usually UI-agnostic and so can be built for UIQ and S60, whereas UI and UI Data MTMs are closely tied to the UI. Therefore it's common to separate the client MTM into a separate DLL from the UI and UI Data MTMs.
- The reason for separate UI and UI Data MTM DLLs is that the UI Data MTM is designed to be lightweight and should not link to any more binaries than necessary.

Drawbacks of the current messaging architecture

The current messaging architecture does not cater well for developers creating message types similar to an existing set of MTMs. For example, when creating a set of MTMs to implement push email, it would be good to plugin a push email transport and reuse the existing email editor and viewer. However, typically editors and viewers are hard-coded to handle certain message types and, short of modifying the type of messages on the fly, it's not possible to achieve a high degree of reuse. In practice this means a developer will have to write four MTM DLLs as well as an editor/viewer for the new message type.

Creating a well-integrated MTM is larger than the scope of this chapter, but the example MTM provides a good starting point.

9.7 The Flickr MTM

This section deals with the task of creating a new MTM. It provides a very specific example of extending the messaging functionality to add an item to the SendAs service, and hence to the UI Send dialogs.

Background on the Flickr service and implementation of the flick upload DLL can be found in Chapter 11.

9.7.1 Goal

The high-level goal of this example is to add a ‘to Flickr’ item to the standard Send options in the S60 and UIQ Send dialogs. This allows users to upload images to their Flickr account directly from a file manager, image gallery or any other application which uses SendUI.

9.7.2 Testing

The MTM can be tested by using the SendWorkbench application to verify that ‘to Flickr’ appears in the Send menu when the data being sent can be represented as an attachment. Files that do not contain images that are passed to the Flickr MTM will fail with KErrNotSupported.

Implementation

The example introduces a new MTM which supports uploading images to Flickr. The MTM is restricted to send only, and has just enough functionality implemented to meet the goal. The UI for the MTM is limited to an ‘uploading to Flickr’ progress report.

9.7.3 Overview

Every MTM should derive at least four classes from the relevant Symbian MTM base classes. Since there is a lot of repetitive framework code, it’s easier to start with an existing example MTM, such as the TextMTM example provided by Symbian, and modify it. Currently the TextMTM is not shipped as part of S60 3rd edition SDK, but is distributed in the UIQ3.0 SDK, in the folder `\Symbian\UIQ3SDK\Examples\SymbianNotCompatible\Messaging\TextMTM`. The reason this is in the ‘not compatible’ directory of the SDK is because the UI elements of the TextMTM example are coded for Symbian’s reference UI – Techview – rather than UIQ.

Figure 9.7 shows how the classes in the Flickr MTM are built up.

The role of each of these classes is to interpret and delegate requests to something which makes sense for the message type.

The UI Data MTM ensures that the MTM is reporting the correct capabilities, the UI MTM ensures that edit and send operations do the right thing, the client MTM takes care of storing the message and the server MTM acts as a message transport.

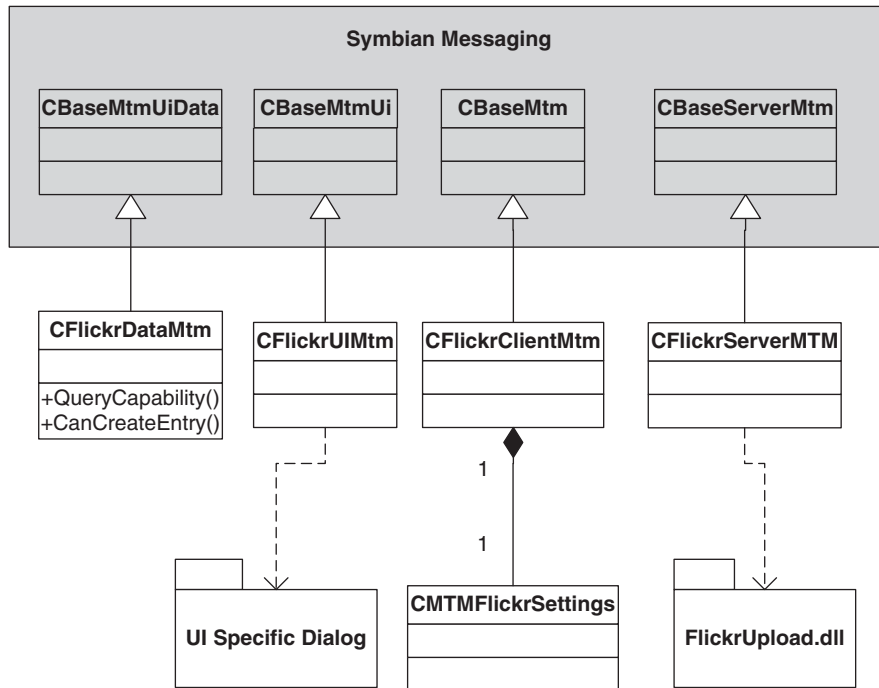


Figure 9.7 Derivation of the Flickr MTM classes from the messaging framework

MTM component	Role
CFlickrMtmUiData	Report capabilities, set the localizable 'To Flickr' text in the send menu
CFlickrMtmUi	Handle message editing and progress using UI-specific dialogs In this example, message editing is mutated into a Send operation with a progress dialog
CFlickrMtmClient	Handle message creation and validation, delegate sending tasks to the server MTM
CFlickrMtmServer	Supports one operation, the sending to Flickr using the FlickrUpload.dll.

9.8 The Flickr Data MTM

9.8.1 Capabilities

Not to be confused with platsec capabilities, the MTM architecture provides a system for enquiring about MTM capabilities at runtime.

The services provided by an MTM are not fixed. There are some standard services such as message creation, deletion and subscription, but also a large number of other services, the availability of which are indicated through the capability mechanism.

The UI Data MTM provides lightweight capability checking.—this allows services such as SendAs to quickly eliminate MTMs that cannot send, for example.

Each MTM component has a method called `QueryCapability()` which can be called to check if an MTM supports a service, or to query simple parameters such as the maximum size of a message supported by the MTM.

The FlickrMTM supports creating new messages (this is obviously essential to support the SendAs service!), supports sending the message and supports attachments. This is represented as follows;

```
TInt CFlickrMtmUiData::QueryCapability(TUId aCapability, TInt& aResponse)
{
    const
    // Query for capability
    {
        switch (aCapability.iUid)
        {
            case KUidMtmQueryCanCreateNewMsgValue:
            case KUidMtmQueryCanSendMsgValue:
            case KUidMtmQuerySupportAttachmentsValue:
                aResponse=ETrue;
                break;
            default:
                return KErrNotSupported;
        }
    }
    return KErrNone;
}
```

SendUI also queries `CanCreateEntryL()` to check if the entry can be created in a specific context. For example that given a specific messaging folder, `CanCreateEntryL()` will be called to test if a child entry be created using this MTM. Please note that although the return type is `TBool`, there is a bug in some UI implementations which expect this to return `KErrNone` to indicate success; returning `ETrue` on S60 means that this MTM will never show up on the Send menu.

```
TBool CFlickrMtmUiData::CanCreateEntryL(const TMsVEntry& aParent,
                                         TMsVEntry& aNewEntry, TInt& aReasonResourceId) const
{
    {
```

```

__ASSERT_ALWAYS(aNewEntry.iMtm== KUidMsgTypeFlickr,
    Panic(EFlickrMtmDataWrongMtm));
__ASSERT_ALWAYS(aNewEntry.iType!=KUidMsvAttachmentEntry,
    Panic(EFlickrMtmDataAttachmentsNotSupported));

aReasonResourceId=0;
// --- Can create services if they are off root ---
if (aNewEntry.iType == KUidMsvServiceEntry)
{
    return (aParent.Id() == KMSvRootIndexEntryIdValue) ? KErrNone :
        KErrNotSupported;
}

// --- Can create messages in local folders ---
if (aNewEntry.iType == KUidMsvMessageEntry)
{
    return (aParent.iMtm.iUid == KMSvLocalServiceIndexEntryIdValue) ?
        KErrNone : KErrNotSupported;
}

return KErrNotSupported;
}

```

An MTM UID is stamped on each message entry in the message store (including service settings), and the corresponding MTM is always consulted before an action is performed. When creating or editing an entry, the MTM should check to see if it is a service entry or a message entry and perform the correct action.

When SendAs or the Send dialogs scan the capabilities of each MTM to see if they are appropriate for the Send menu, the capabilities are queried in the order listed in the table below. The expected response must match the table or the MTM will not be listed.

Function	Expected response
QueryCapability() called with KUidMtmQueryCanCreateNewMsgValue	ETrue
CanCreateEntryL()	KErrNone
QueryCapability() called with KUidMtmQueryMaxBodySizeValue	Body text size. If the bearer has no size limit then choose a suitably large number such as KMaxTInt
QueryCapability() called with KUidMtmQueryMaxTotalMsgSizeValue	Total size limit including attachments, again set to KMaxTInt if there is no inherent limit

9.8.2 Specifying the Send menu text

The default text used in the Send menu is the human readable name for the MTM as specified in the registration resource file. This is fine for initial development purposes, but a localizable string is required for the final version of the code.

This is achieved by supplying a function with the flag `EMtudCommandSendAs` in the UI Data MTM resource file and ensuring that `functiontext` is localized.

```
// MTM-specific function array
RESOURCE MTUD_FUNCTION_ARRAY r_textud_function_array
{
    functions=
    {
        MTUD_FUNCTION { functiontext="to Flickr";
                        command=KMtmUiFunctionSendAs; flags=EMtudCommandSendAs;}
    };
}
```

9.9 The Flickr UI MTM

The UI MTM often contains the bulk of the implementation of the four MTMs. This is because the UI MTM does a lot of decision making, may supply a UI for reporting the progress of the client/server MTMs and must provide a way of editing the message and the service settings.

For complex message types, the editing is usually delegated to a viewer/editor application. The viewer/editor has the hidden attribute set in the AIF file so that it doesn't show up in the application menu. The viewer/editor is passed the `TMsvId` of the newly created message as part of the command line parameters.

The FlickrMTM has no editor, it simply puts up a dialog and starts uploading the message to the Flickr website.

9.9.1 Editing a message

```
CMsvOperation* CFlickrMtmUi::EditL(TRequestStatus& aStatus)
// Edit
{
    TUid type = iBaseMtm.Entry().Entry().iType;
    iServiceId = iBaseMtm.Entry().Entry().iServiceId;

    __ASSERT_DEBUG(type==KUidMsvMessageEntry ||
                  type==KUidMsvServiceEntry,
                  Panic(EFlickrMtmUiWrongEntryType));
```

```

if ( type == KUidMsvMessageEntry )
{
    return MessageEditL(aStatus);
}
else
{
    return ServiceEditL(aStatus);
}
}

```

You should check to see if the message is a service or message type in order to perform the correct action. The Flickr MTM does not support editing of services.

Editing the message causes a `CImageUploadOperation` to be kicked off. This is essentially a send operation wrapped up in an ‘Uploading to flickr’ dialog in order to give the user some feedback.

```

CMsvOperation* CFlickrMtmUi::MessageEditL(TRequestStatus& aStatus)
// Message editing
{
    aStatus = KRequestPending;

    CMsvEntry& entry = iBaseMtm.Entry();

    CImageUploadOperation* op = new(ELeave)CImageUploadOperation(*this,
        Session(), entry.Entry().Id() , CActive::EPriorityStandard,
        aStatus);
    CleanupStack::PushL(op);
    op->StartL();
    CleanupStack::Pop(op);
    return op;
}

```

9.10 Flickr Client MTM

9.10.1 Message layout

A client MTM dictates how the message store and message entries are used to hold messages. A single message seen by the user in the inbox can be composed of many child message entries that store the structure. However, in practice most messages are represented by a single message entry.

The default behavior for an MTM which supports attachments is that requests from the clients to add attachment files will be handled automatically by the attachment manager. The attachment manager associates files with a particular message entry, and allows the MTM to enumerate and access the message.

Potentially, a Flickr message type could contain further information such as body text containing description, tags and a privacy tag (these are features of the Flickr API), but for this example the Flickr MTM message payload can be defined entirely as a single attachment. Only the filename and the image data are used, which leads to a very simple message representation.

When an MTM is asked to add an attachment, the default behavior (if not overridden by the MTM) is to store the attachment as part of the message store. The default is acceptable for the Flickr MTM.

When a message needs to be sent, the server MTM uses the attachment manager API (`MMsvAttachmentManager`) to retrieve the attachments.

Handling a SendAs request

Once a message has been created, a SendAs client may request the message be sent immediately. This is actioned by a call to the client MTM's `InvokeAsyncFunctionL()` with a function `UID KMTMStandardFunctionsSendMessage`.

```
CMsvOperation* CFlickrMtmClient::InvokeAsyncFunctionL(TInt aFunctionId,
    const CMsvEntrySelection& aSelection, TDes8& aParameter,
    TRequestStatus& aCompletionStatus)
// Call MTM-specific operation asynchronously
{
    CMsvOperation* operation = NULL;
    switch (aFunctionId)
    {
        case KMTMStandardFunctionsSendMessage:
        {
            operation = (Entry().Session().TransferCommandL(aSelection,
                aFunctionId, aParameter, aCompletionStatus));
        }
        break;
        default:
            __ASSERT_DEBUG(EFalse, gPanic(EFlickrMtmClientCommandNotSupported));
            break;
    }
    return operation;
}
```

The client MTM must return an object derived from `CMsvOperation` – despite the return type, it must not return `NULL`.

9.10.2 Transferring commands from client to server

The Flickr MTM client MTM delegates the SendAs call to the server MTM.

The message server provides a convenient `TransferCommandL()` to perform the delegation in which a client side `CMsvOperation` representing the server-side operation is created and acts as a proxy to allow

progress information to flow back from server to the client. The server operation may be cancelled at any time by calling `Cancel()` on the client side `CMSvOperation`.

After returning the `CMSvOperation` from `TransferCommandL()`, the client MTM has nothing further to do with the transaction and ownership of the operation is transferred to the caller of `InvokeAsyncFunctionL()`.

9.11 The Flickr Server MTM

As shown in Figure 9.6, the server MTM runs inside the message server process, on the other side of the client/server boundary from the client MTM. When a `TransferCommandL()` is called by the client MTM, the server MTM is invoked via `StartCommandL()` with parameters from the client and a request status which must be completed when the server MTM has finished its task.

The FlickrMTM only supports one command, `KMTMStandardFunctionsSendMessage`.

```
void CFlickrServerMtm::StartCommandL(CMSvEntrySelection& aSelection,
    TInt aCommand, const TDesC8& /*aParameter*/, TRequestStatus&
    aStatus)
    // Run MTM-specific command on selection of entries
    // Only command supported is Refresh
{
    AssertIdle(); // Standard check to assert that we are not already busy

    switch (aCommand)
    {
        case KMTMStandardFunctionsSendMessage:
        {
            SendToFlickrL(aSelection[0], aStatus);
        }
        break;

    }
}
```

9.11.1 The message transport

In general the server MTM provides the message transport – the code which actually sends the message over the underlying transport, such as Bluetooth, TCP/IP or another protocol.

At the point the message server gets hold of the message, it will be stored in the message store by the client side code as a mix of `TMsvEntry` flags, rich text and/or attachments.

Essentially a message transport is responsible for translating or parsing the message from the format in which it is stored in the message store

into one that can be transmitted. The transport may also change the state of the message, or perhaps delete it after it has been successfully sent.

For the Flickr MTM, the server MTM needs to extract the attachment from the message store and upload the image using the FlickrUpload.dll from Chapter 11, then delete the message from the outbox.

The following code fragment shows how the message is retrieved from the `TMsvId` passed from the client MTM and demonstrates how the attachment is retrieved from the message store.

```
void CFlickrServerMtm::SendToFlickrL(TMsvId aMessage, TRequestStatus&
    aStatus )
{
    iCurrentOperation = EFlickrSendPhoto;
    iReportStatus = &aStatus;
    aStatus = KRequestPending;

    iServerEntry->SetEntry(aMessage);

    CMsvStore* store = iServerEntry->ReadStoreL();
    CleanupStack::PushL(store);
    MMSvAttachmentManager &attachMan = store->AttachmentManagerL();

    if( attachMan.AttachmentCount() != 1 )
    {
        User::Leave(KErrNotSupported);
    }

    CMsvAttachment* attachMent = attachMan.GetAttachmentInfoL(0);
    CleanupStack::PushL(attachMent);
    const TDesC8& mimeType = attachMent->MimeType();

    iFlickrRestAPI->IssueImagePostL(attachMent->FilePath());

    CleanupStack::PopAndDestroy(2,store);
}
```

Since the code is running in the context of the message server process, it has direct access to the attachment file in the message store, or in the case of a linked file the file path will point to a public folder, such as `c:\data\images` in our earlier example.

Operations running in a system server such as the message server thread must be cooperative. Operations should be kept short and asynchronous in order for other messaging requests to be serviced in good time.

`CFlickrRestAPI::IssueImagePostL()` is an asynchronous call which makes a callback to `FlickrRestOperationComplete()` when complete.

The instance variable `iReportStatus` keeps track of the client-side request status passed in from the client MTM and is completed on completion of the image upload.

```
void
CFlickrServerMtm::FlickrRestOperationComplete
(MFlickrRestAPIObserver::TOperationType aOpType, TInt aError)
{
    User::RequestComplete(iReportStatus, aError);
    TMsgId id = iServerEntry->Entry().Id();

    iServerEntry->SetEntry(iServerEntry->Entry().Parent());
    iServerEntry->DeleteEntry(id);
}
```

On completion of the upload operation, the message is deleted from the store. The Flickr MTM has no notion of retry and so the message must be deleted – the user must try to send the image again.

An entry can only be deleted from its parent, so `iServerEntry` is set to the parent and the message deleted.

Reporting progress

The messaging architecture provides for polling of progress on an asynchronous object and final status of an operation.

Most messaging operations are asynchronous and, following standard practice, the value in the `TRequestStatus` on completion indicates success or failure.

A single integer is rarely enough to describe the full context of the error and so the messaging APIs provide a way of passing intermediate progress for an operation. All client-side messaging operations derive from `CMsvOperation`, which encapsulates support for reporting progress.

When a client requests progress on a pending `CMsvOperation`, an 8-bit package buffer is copied from the server side MTM to the client. This package contains progress information in a format understood by both the server and client MTMs, but opaque to the message server.

The client calls

```
virtual const TDesC8& CMsvOperation::ProgressL()=0;
```

And the server MTM supplies

```
virtual const TDesC8 &Progress()=0;
```

The common format between client and server MTMs is usually achieved by defining the progress data structures in a shared header file.

A type-safe package buffer is defined to pack and unpack the progress structure as an 8-bit descriptor. The package buffer has a maximum size limit of 255 bytes – larger progress buffers will cause an ‘MSGS Client 11’ panic.

Here is the progress structure used for the Flickr MTM:

```
class TImageUploadProgress
{
public:
    TInt iErrorCode;
    TBuf8<150>iErrorMessage;
};

typedef TPckgBuf<TImageUploadProgress> TImageUploadProgressBuf;
```

The client polls for progress periodically and so the server operation should keep the progress structure valid and up to date, in the knowledge that the message server may copy the structure to the client at any time.

The progress structure should contain any information that charts progress and could be useful for the UI to display; it should also contain context about any error conditions encountered during the operation.

Any buffer used by the client for receiving progress needs to remain in scope for the duration of the asynchronous operation – you should therefore store progress structures as a member of the CActive-derived class making the request rather than using a local variable, as the local variable will be out of scope before the asynchronous operation completes. Getting this wrong is a common source of hard-to-diagnose bugs in asynchronous programming, as the point at which the memory formerly used by any stack-based buffer is overwritten, is timing dependent.

9.12 MTM DLLs and Platsec

The required platsec capabilities differ between UI platforms. In general the DLLs containing the client, UI and UI Data MTMs may be loaded into any application using messaging services and so are similar to any DLL providing a platform-wide service. In the case where the MTMs are only designed to implement a new SendAs message type, they can have a more restricted set of capabilities in UI platforms that implement the architecture shown in Figure 9.4. Today this includes UIQ but not S60.

However, expecting DLLs to be signed with All-TCB is unreasonable, so instead our recommendation is to test with a variety of applications on a device to discover a reasonable capability set for the client-side MTMs.

The ‘private’ MTM component, that is the server MTM, requires capabilities equal to the loading process – the messaging server process. Your server-side MTM component should be given the following capabilities:

```
NetworkServices, LocalServices, ReadUserData, ReadDeviceData,
WriteDeviceData, ProtServ, NetworkControl
```

Of these, `ReadDeviceData`, `WriteDeviceData`, `ProtServ` and `NetworkControl` are all from the extended capability set – therefore it is likely that additional information will need to be provided when submitting the MTM suite to any signing process stating the reasons for requiring these capabilities – namely that you are providing messaging server plugins.

An MTM going through the signing process will need all UID3s for EXEs and DLLs to be assigned from the protected UID range ($0 \times 200000000 - 0 \times 2FFFFFFF$) – UIDs from this range can be obtained via the Symbian Signed website (www.symbiansigned.com).

9.13 FlickrMTM Shared Settings

As discussed in the section ‘Reporting progress’, some data structures need to be shared between the four MTM layers.

Usually a utility DLL is created in order to store any shared constants, data structures and common utility functions.

The Flickr MTM uses `FlickrMtmsettings.dll` to store message type UIDs, constants, operation IDs and progress structures. The DLL contains functions to set and retrieve the default service. In a more complex MTM, the shared DLL may serve to share code between the client and server for saving and retrieving messages from the store.

9.14 Installation of an MTM

Writers of MTMs need to inform the message server of the properties of the MTM by supplying a registration file. This is a compiled resource file placed in the `\resource\messaging\mtm\` folder.

In versions of Symbian OS before v9, the resource file is placed in `\system\mtm\`.

The resource file should define the MTM’s properties using the resource structures `MTM_INFO_FILE`, `MTM_CAPABILITIES`, and from v9.0, `MTM_SECURITY_CAPABILITY_SET`.

When installing a new set of MTMs on a phone, you must take the additional step of calling `CMsgvSession::InstallMtmGroup()` to notify the message server that a new MTM is available. When this is done, all running message server client processes are also notified that a new MTM is available. Typically you would do this by running an additional program at the end of the installation process.

Uninstallation also requires running a simple exe to unregister the MTM by making a call to `CMSv Session::De Install Mtm Group`. Running clients are then dynamically notified that the MTM is no longer available.

```
RESOURCE MTM_INFO_FILE
{
    mtm_type_uid = 0x20009972; // Specifically allocated for the Flickr MTM
    technology_type_uid = 0x20009972;
    components =
    {
MTM_COMPONENT_V2
    {
        human_readable_name = "Flickr";
        component_uid = KUidMtmServerComponentVal;
        entry_point = 1;
        version = VERSION_V2 {};
        filename = "FlickrMtmServer.dll";
    },
MTM_COMPONENT_V2
    {
        human_readable_name = "Flickr";
        component_uid = KUidMtmClientComponentVal;
        entry_point = 1;
        version = VERSION_V2 {};
        filename = "FlickrMtmClient.dll";
    },
MTM_COMPONENT_V2
    {
        human_readable_name = "Flickr";
        component_uid = KUidMtmUiComponentVal;
        entry_point = 1;
        version = VERSION_V2 {};
        filename = "FlickrMtmUi.dll";
    },
MTM_COMPONENT_V2
    {
        human_readable_name = "Flickr";
        component_uid = KUidMtmUiDataComponentVal;
        entry_point = 1;
        version = VERSION_V2 {};
        filename = "FlickrMtmUiData.dll";
    }
    };
}

// flags to indicate that can send messages, and handle body text
RESOURCE MTM_CAPABILITIES
{
    send_capability = 1;
    body_capability = 0;
}

// Additional capabilities required by clients of the
// flickr MTM.
// Since we are sending data over an IP network, we
// require the client to have the same capabilities as
```

```
// if they were going to do it themselves
RESOURCE MTM_SECURITY_CAPABILITY_SET
{
    capabilities = {ECapabilityNetworkServices};
}
```

9.15 Summary

In this chapter we have learnt how to:

- use the SendAs API to send messages in a bearer-independent way from our application
- use the send dialogs provided by the UI platforms in the case where we want to offer the user a choice of bearer over which to send the message
- extend the SendAs functionality by implementing a new set of MTMs, in this case to allow the upload of photos to Flickr.

10

OBEX

This chapter starts with an introduction to the OBEX protocol – not being a mainstream protocol, we go into a fair amount of depth to give you a feel for the feature set of OBEX. After this, we explore the APIs offered by the Symbian OS OBEX implementation. We also illustrate the use of these APIs using examples drawn from an example implementation of a Bluetooth FTP server.¹

The chapter also shows how the OBEX API has developed from its origins in ER5 to the current implementation in Symbian OS v9.1, and highlights the most recent changes for Symbian OS v9.2.

10.1 OBEX Overview

The OBEX (or OBject EXchange) protocol was originally developed as part of the Infrared Data Association's infrared protocol suite. It has since been adopted by the Bluetooth Special Interest Group as the basis of the Generic Object Exchange Profile (GOEP) family of Bluetooth profiles, and by the USB Implementers' Forum as part of the Wireless Mobile Communications Device Class (WMCDC) specification. It is intended to provide an easy to use, lightweight and flexible means of transferring 'objects' from device to device, where an object can be almost any conceivable data entity from a vCard to a multi-megabyte media file.

OBEX performs a very similar function to HTTP, and both have very similar features at a high level. However, OBEX is targeted at devices with limited resources, so it omits some of HTTP's functionality in order

¹ Two things are worth noting here – firstly, whilst this example application is a good demonstration of what you can do with OBEX, Bluetooth FTP servers are already implemented in shipping devices.

Secondly, the implementation has not qualified by the Bluetooth SIG, but has been tested using a Windows-based Bluetooth FTP client, running USB TDK Bluetooth adapter and Ezurio Bluetooth Software 1.4.2 Build 18.

to focus on providing a service that can be implemented relatively easily on resource-constrained devices. Concentrating on similarities for a moment – both protocols transfer arbitrary data in a request–response fashion, both use the concept of ‘headers’ to describe that content and ‘body’ to contain the content itself. Both have the concept of PUTting and GETting objects.

However, OBEX differs in some key areas – it is a binary-based protocol rather than a text-based protocol like HTTP. Unlike HTTP, which uses cookies to maintain session state (as the protocol is inherently stateless), OBEX handles state at the protocol level. And finally, whilst HTTP is most often run over TCP (although other bindings do exist), the most popular bindings for OBEX are Bluetooth RFCOMM, IrDA TinyTP and USB.

The following sections give a brief overview of OBEX itself, before going into the specifics of the Symbian implementation. The OBEX specification is available (for a fee) from www.irda.org.

10.1.1 OBEX Fundamentals

The OBEX protocol is asymmetric, meaning that the two entities involved in an OBEX transfer play different roles – OBEX *client* and OBEX *server*. Whilst connected by a session, an OBEX client and server are termed as OBEX *peers*. The terms client and server are used here in the traditional sense that the client makes *requests* of the server, and the server provides a *response* to these requests.

In OSI model terms, OBEX belongs in the session layer. It defines a structured dialog where a client sends *request packets* to a server, after which the server sends *response packets* to the client. This request–response exchange happens in strict rotation; the server may never send a response packet without having first received a request packet, and a client may never send a second request packet without having received a response to its first request.

The formation and sending of request packets begins when the application acting as the OBEX client (known forthwith as the OBEX client application) issues an *OBEX command*. There are surprisingly few standard OBEX commands in version 1.2 of the OBEX specification (the version supported by Symbian OS):

- **Connect** – establish an OBEX session
- **Disconnect** – close the OBEX session
- **Put** – transfer an object from the OBEX client to the OBEX server
- **Get** – transfer an object from the OBEX server to the OBEX client
- **SetPath** – alter the path on the OBEX server
- **Abort** – abandon the current OBEX command

A Connect, Put or Get command may result in multiple request–response exchanges between the OBEX client and server before the command can be said to be complete. Mechanisms are provided within the protocol to allow the client and server to determine when a particular command is completed. The Disconnect, SetPath and Abort commands are always dealt with in a single request–response exchange.

Figure 10.1 shows an example of how an individual OBEX command (in this case a Put) translates into multiple request–response exchanges. Please note that the application/OBEX implementation split is Symbian OS-based, although the traffic ‘on the air’ would follow a similar pattern regardless of the OS in use.

With such a compact command set, it may be appreciated that the flexibility of OBEX is mainly a result of the rich model provided for describing the transferred objects themselves. The OBEX object model uses *headers* to describe these objects.

10.1.2 OBEX Headers

In the OBEX protocol, most information is conveyed as headers. There are a large number of different possible header types, but all of these header types can be split broadly into two groups: headers used to represent objects to be transferred (these belong to the ‘object model’) and headers used by the OBEX protocol itself for its own purposes, such as authentication and the selection of directed services.

When present in an OBEX packet, each header consists of one byte of header information (HI) which identifies the header type and encoding, followed by multiple bytes of header value (HV) which encodes the actual value of the header. There are four different encoding types that

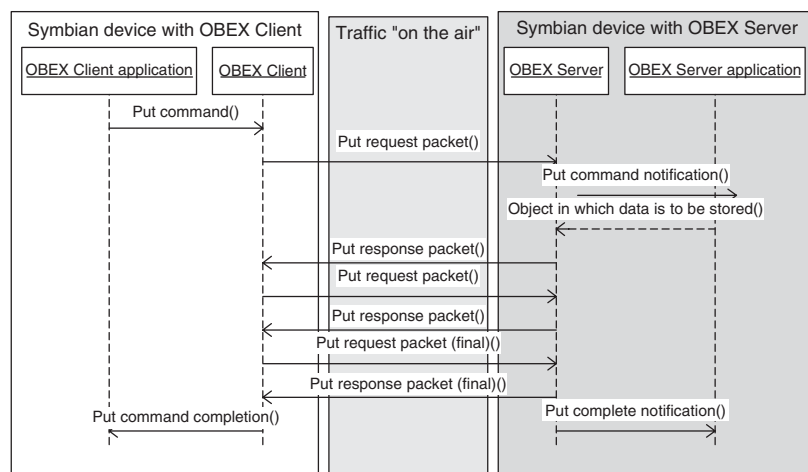


Figure 10.1 Simplified view of OBEX Put command processing in Symbian OS, showing multiple packet exchanges for a single command

can be used for a header: a length-prefixed null terminated Unicode string; a length-prefixed byte sequence; a one-byte quantity; and a four-byte quantity. The encoding type for a header is held in the upper two bits of the HI, with the lower six bits of the HI identifying the header type. Some examples are show in Table 10.1.

Headers used in the object model

The object model describes how OBEX represents objects. An object is represented as a collection of associated headers. During an object transfer, the main payload or ‘body’ of an object is represented by one or more body headers. There may be any amount of metadata associated with an object – for such metadata, there are specialized headers types such as Name, Type and Date.

The headers included in a particular transfer, and the way in which the headers are interpreted, will depend largely on the nature of the service being used by the OBEX client and provided by the OBEX server. For instance, if a file is being transferred to another device for storage, the Name header will be of primary importance to convey the file name. If the file is being transferred purely to be played (e.g., a JPEG file being sent to a media player for display), the Type header might be of greater importance.

A few important object model header types are listed below. More details on these and other headers can be found in the OBEX specification.

Body and end-of-body headers The main payload of an object can represent almost anything: the contents of a file, a contact, a piece of XML, the result of a database query, etc. When transferred using OBEX, the payload usually (unless it is very small) has to be fragmented to allow it be encoded into OBEX packets, which have a finite capacity. The payload is fragmented into zero or more Body headers and/or an End-of-Body header, and sent in ordered sequence. These headers are reassembled in the same sequence by the receiving OBEX client or server.

The Body header and End-of-Body header perform almost identical tasks, and both header types can carry payload data. The End-of-Body

Table 10.1 Some common header types and their HI values

HI value	Header type	Encoding	Usage
0x01	Name	Null terminated Unicode	Name of the object
0x42	Type	Byte sequence	Describes media type of object
0xC3	Length	Four-byte quantity	Describes number of bytes in object payload

header differs from the Body header only because it has a different HI value, which indicates that it contains the final piece of the object's payload. As a result, while the payload may be split into a number of Body headers, only a single End-of-Body header is allowed.

For objects with no payload, the End-of-Body header may be included or omitted from a transfer. For some services such as file transfer, however, the presence of an empty End-of-Body header has a distinct meaning from the absence of any End-of-Body header at all (e.g., the difference between an empty file and no file at all).

Name The Name header represents the name of the object being transferred. As always, the interpretation of the name will depend on the OBEX service being used; for an OBEX file transfer for example, the Name header will be used to represent the file name.

Type The Type header represents the object type being transferred. Usually, the Type header contains an IANA registered media type (e.g., 'text/html') and is used by the receiving OBEX client or server to determine the action to be taken to deal with the received object. Depending on the OBEX service being used, the Type header may be unnecessary or optional – the service may well determine what the object type is by other means.

Length If the number of bytes in an object is known before a transfer begins, the Length header may optionally be used to represent that number. The header is transferred early in the object exchange as it provides useful data for the receiver – it can be compared with available storage space to determine whether the transfer should be allowed to continue, and it can be used during a transfer to determine what percentage of an object has been transferred so far. It should also be noted that the Length header is not guaranteed to be an absolutely accurate size of the object; it is intended for use in situations such as displaying a progress bar where exact numbers are not necessary.

HTTP To allow reuse of some HTTP-based features, the OBEX specification allows HTTP headers to be represented and transferred as part of an object.

Headers used by the OBEX protocol

As well as those headers used in the object model, there are additional headers used internally by OBEX to control various aspects of the protocol.

Authentication challenge and authentication response headers The authentication challenge and authentication response headers provide the OBEX protocol with the ability to perform password-based authentication during an exchange. This authentication is quite flexible:

- the OBEX client can authenticate the OBEX server
- the OBEX server can authenticate the OBEX client
- both the OBEX client and the OBEX server can authenticate each other.

Or, of course, authentication can be missed entirely, for instance where the underlying transport provides inbuilt security (e.g., Bluetooth pairing or IrDA's requirement for line of sight).

Although the OBEX specification allows authentication to be performed for any command, in practice it is most commonly performed during the Connect command used to establish an OBEX session.

Target, Who and Connection ID headers The OBEX specification defines the concept of a *directed connection*. This is a means by which an OBEX client can be routed and connected to an OBEX server providing the particular service the client wants. This is an optional feature, most useful when many OBEX services have been multiplexed over a single underlying transport connection. The Target, Who and Connection ID headers all play a role in the creation of directed connections.

- The Target header is sent by the OBEX client during a directed connection attempt to identify the service it wishes to connect to.
- The Who header is sent by the OBEX server during a directed connection attempt to identify the service it is offering.
- The OBEX server provides the OBEX client with a Connection ID header value when it accepts the OBEX client's connection to form a session.
- The same Connection ID header is used by the connected OBEX client to identify its session to the OBEX server during subsequent commands.

Figure 10.2 illustrates this sequence.

Please note: using directed connections is often unnecessary since the transports underlying OBEX offer their own means of differentiating and selecting services (Bluetooth has SDP, IrDA has IAS) and their own multiplexing and routing capabilities. The subject of finding and connecting to OBEX services in this way will be discussed in section 10.1.5.

10.1.3 OBEX Packets

To perform an OBEX transfer, the various headers described above have to be placed into and (following transmission) extracted from OBEX packets.

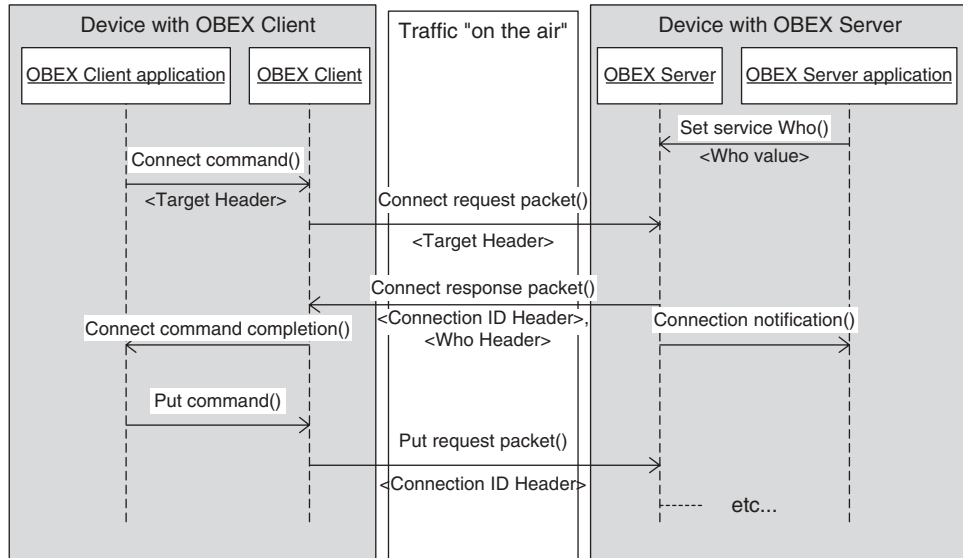


Figure 10.2 (Symbian OS-based) example of use of Target, Connection ID and Who header in directed connections

The OBEX specification defines formats for a number of different packet types. They can be divided into request packet types, which are sent from the client to the server; and response packet types, which are sent from the server to the client.

Request packet format

All request packets have the format shown in Table 10.2.

The opcode field indicates the type of request that the OBEX client is requesting from the OBEX server. The OBEX specification defines an opcode for each of the six OBEX commands (Connect, Disconnect, Put, Get, SetPath, Abort). The most significant bit of the opcode field is termed the 'final' bit, which is used in a command-specific way to signal the end of the current command, or the end of a phase of the current command.

The packet length indicates the total length of this packet, including the opcode field and the length field itself. As a consequence, the minimum length possible is three bytes, and the maximum length possible is 65,535 bytes ($2^{16} - 1$). Other factors may limit the usable packet size, and this is discussed later in this section, and in section 10.2.5).

Table 10.2 OBEX request packet format

Byte 0	Bytes 1–2	Bytes 3–n
Opcode	Packet length	Additional request data and/or headers

The final part of the request packet contains any additional request data required by the particular OBEX command, and the headers themselves. In practice, only the Connect and SetPath commands require additional request data.

- For the Connect command, additional request data is used to communicate the OBEX version number, a flag byte and most importantly the OBEX packet length that can be received by the client.
- For the SetPath command, additional request data is used to communicate flags related to navigation and creation of directories on the OBEX server.

Response packet format

All response packets have the format shown in Table 10.3.

The response code field provides a means by which the OBEX server can communicate the outcome of the most recently received request from the OBEX client. There is a wide range of codes defined in the OBEX specification (based on HTTP status codes). Some represent successful outcomes (e.g., ‘Success’, ‘Created’, ‘Accepted’), some represent unsuccessful outcomes (e.g., ‘Bad Request’, ‘Forbidden’, ‘Unauthorized’), and some are informational and may indicate success or failure depending on context (e.g., ‘Partial Content’). Again, the most significant bit of the response code field is termed the ‘final’ bit, and is used in a command-specific way to signal the end of the current command, or the end of a phase of the current command.

The meaning of the packet length field is identical to that of the packet length field in the request packet.

The final part of the response packet contains any additional response data required by the current OBEX command, and the headers themselves. In practice, only the Connect response contains additional response data – it is used to communicate the OBEX version number, a flag byte and most importantly the maximum OBEX packet length that can be received by the server.

Packet sizes

As mentioned above, although the maximum packet size is limited to $2^{16} - 1$ by the size of the packet length field, other factors can affect the

Table 10.3 OBEX response packet format

Byte 0	Bytes 1–2	Bytes 3–n
Response code	Packet length	Additional response data and/or headers

usable size of an OBEX packet, since the actual maximum packet size is not known until the Connect command is complete.

The most common factor is the capability of the peer OBEX device. Some devices or implementations of OBEX may have limited buffer space into which an OBEX packet can be received. During processing of a Connect command, the OBEX client and server exchange details of the maximum OBEX packet length they can each receive. If the connection is accepted, the OBEX client must respect the maximum packet length of the OBEX server, and the OBEX server must respect the maximum packet length of the OBEX client.

The minimum allowable value for the maximum packet size is 255 bytes; this is also the default maximum packet size used when processing a Connect command, before the maximum packet size of the remote device is known.

In practice, the packet size tends only to affect the performance of an OBEX transfer. If the allowed packet size is very small, however, it may prevent some headers from being transferred entirely – headers may not be fragmented between OBEX packets.²

Header encoding

Headers may be encoded into the space in a packet following the opcode or response code field, packet length field and additional request or response data. An OBEX client or server attempting to transfer an object will attempt to use as much of the available space in a packet as possible to transfer headers (assuming there are headers to transfer).

Because the size of the OBEX packets is limited, not all headers will fit into one request packet and so must be sent in consecutive packets belonging to the same command. There are few rules governing the order in which headers are sent, but generally the headers will be prioritized as follows:

1. Non-object model headers
2. Object model headers other than Body/End of Body
3. Body/End-of-Body headers.

10.1.4 OBEX Session Protocol

As mentioned above, the OBEX session protocol defines a structured dialog where a client sends *request packets* to a server, after which the server sends *response packets* to the client. This exchange proceeds in strict rotation, request–response, request–response until the required

² Strictly speaking, even Body headers are not fragmented between OBEX packets; rather, the payload of an object is fragmented into multiple individual Body headers.

transfer or transfers are complete. The following sections summarize the packet exchanges involved for each OBEX command.

Connect

The Connect command is used by OBEX client application to establish a session with an OBEX server. The Connect command must therefore be used before any other OBEX commands for a given session.³

A Connect command will comprise either a single request–response exchange, or two request–response exchanges, depending on whether the client, or server, or both client and server, are requesting authentication. Whether authentication is required or not is an application-specific decision. The various authentication scenarios are described below.

Connect (no authentication) The sequence of events is as follows:

1. The OBEX client sends a Connect request packet to OBEX server.
2. The OBEX server receives the Connect request packet. It then prepares and sends a Connect response packet to the OBEX client.
3. The OBEX client receives the Connect response packet.

Connect (client authenticates server) The sequence of events is as follows:

1. The OBEX client sends a Connect request packet to the OBEX server containing an authenticate challenge header.
2. The OBEX server receives the Connect request packet, and extracts the authenticate challenge header. It then prepares and sends a Connect response packet to the OBEX client. The Connect response packet contains an authenticate response header (preparation of the authentication response header often involves accepting user input of a password).
3. The OBEX client receives the Connect response packet, extracts the authenticate response header and verifies the authenticity of the OBEX server. If the authentication is successful, the OBEX client will proceed with further OBEX commands. If the authentication is not successful, the OBEX client may proceed with a Disconnect command or simply disconnect the transport.

Connect (server authenticates client)

1. The OBEX client sends a Connect request packet to the OBEX server.

³ This is not strictly true; there are variations of OBEX that are not connection oriented (e.g., OBEX over IrDA Ultra), but the vast majority of applications of OBEX use the connection-oriented version.

2. The OBEX server receives the Connect request packet. It then prepares and sends a Connect response packet to the OBEX client with the response code value 'Unauthorized' and an authenticate challenge header originating from the OBEX Server.
3. The OBEX client receives the Connect response packet, and extracts the authenticate challenge header. It then prepares and sends a second Connect request packet similar to its first, but this time including an authenticate response header.
4. The OBEX server receives the Connect request packet, and extracts the authenticate response header and verifies the authenticity of the OBEX client. If the authentication is successful, the OBEX server responds to the OBEX client with a Connect response packet with a response code value 'Success'. If the authentication is unsuccessful, the OBEX server may respond with a Connect response packet containing a response code indicating the failure of the connection.
5. (Assuming successful authentication) the OBEX client receives the Connect response packet, and the OBEX session has been connected.

Connect (both client and server authenticate each other)

1. The OBEX client sends a Connect request packet to the OBEX server containing an authenticate challenge header.
2. The OBEX server receives the Connect request packet. It then prepares and sends a Connect response packet to the OBEX client with the response code value 'Unauthorized' and another authenticate challenge header, this one originating from the OBEX server.
3. The OBEX client receives the Connect response packet, and extracts the authenticate challenge header. It then prepares and sends a second Connect request packet similar to the first (including a copy of the original authentication challenge header), but this time includes an authenticate response header to satisfy the OBEX server's requirements.
4. The OBEX server receives the Connect request packet, and extracts the authenticate response header and verifies the authenticity of the OBEX client. If the authentication is successful, the OBEX server responds to the OBEX client with a Connect response packet with a response code value 'Success', and an authenticate response header of its own to satisfy the OBEX client's requirements. If the authentication is unsuccessful, the OBEX server may respond with a Connect response packet containing a response code indicating the failure of the connection.
5. (Assuming successful authentication of the OBEX client by the OBEX server), the OBEX client receives the Connect response packet, and

extracts the authenticate response from the OBEX server. It then attempts to verify the authenticity of the OBEX server. If this succeeds, the OBEX session has been connected.

Connection failure cases Aside from authentication failures, there are other reasons why a Connect command may fail. For example, some client applications or server applications will only work if connected to a ‘strict peer’ – that means the Target header specified by the OBEX client in a Connect command matches the Who header associated with the OBEX server that processes the Connect command. When such a failure case is encountered, the OBEX client can abandon a Connect command by disconnecting the OBEX transport. The OBEX server can refuse a Connect command in two ways:

- send a response packet with a non-success response code.
- disconnect the OBEX transport.

Disconnect

The Disconnect command is used by the OBEX client application to close down an existing session with an OBEX server. A Disconnect command will always comprise a single request–response exchange. When the OBEX client application performs a Disconnect command, the OBEX client sends a Disconnect request packet. Upon receiving this, the OBEX server can send back a Disconnect response packet (which must always indicate successful disconnection), or can simply disconnect the OBEX transport to complete the command.

Put

The Put command is used by the OBEX client application to transfer an object from the OBEX client to the connected OBEX server. The OBEX client application specifies the object to be sent, and the OBEX client performs the necessary processing to convert this into a sequence of request–response exchanges. A Put command can be completed in a single request–response exchange, but this depends upon the size of the object being transferred and the maximum OBEX packet size that the OBEX client is capable of sending and the OBEX server is capable of receiving.

Most Put commands will require more than one request–response exchange. The OBEX client uses the final bit in the request opcode to indicate the final request packet of a given Put command so that the OBEX server can complete the object transfer (e.g., commit the received data to file) and prepare itself to accept the first request packet of the next command from the OBEX client (see Figure 10.1).

Get

The Get command is used by the OBEX client application to request that the connected OBEX server transfers a specified object to the OBEX client. The Get command is a little more complex than the Put command, as it has two distinct phases.

In the first phase, the OBEX client sends a 'specification object' to the server. In the second phase, the OBEX server sends an object, whose content is dependent on the specification object, back to the OBEX client. The phases of a Get command should not be confused with the request–response exchange of packets; both phases of the Get may consist of several request–response exchanges of packets as illustrated in Figure 10.3.

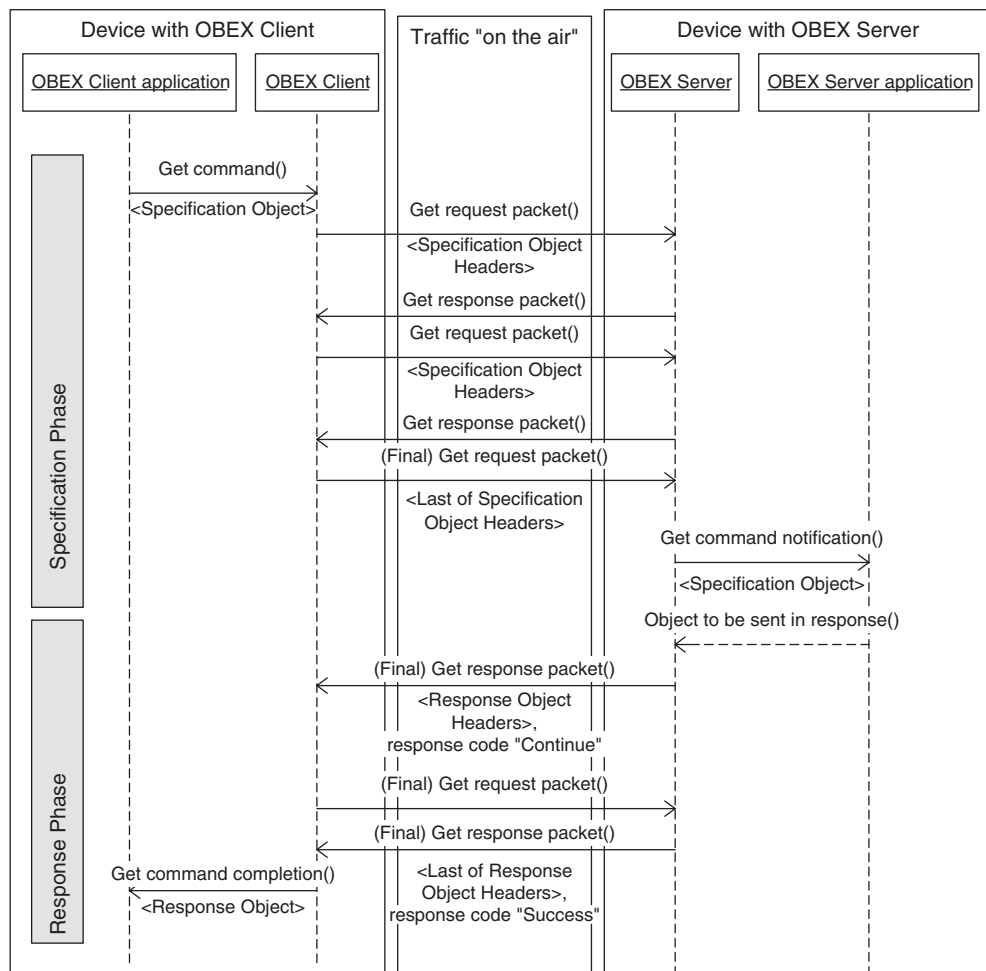


Figure 10.3 (Symbian OS-based) simplified view of OBEX Get command

In the first phase, the data flows predominantly from the OBEX client to the OBEX server in the request packets, which, as always, are acknowledged by the OBEX server returning response packets. In the last request packet to contain specification object data, the OBEX client sets the final bit to indicate that the first phase is over.

Upon receiving the request packet with the final bit set, the OBEX server completes the specification object transfer, and decides on the object to be sent back to the client – this decision is entirely dependent on the service being used by the client and offered by the server. The second phase commences, and the data now flows predominantly from the OBEX server to the OBEX client in the response packets. Effectively, during this phase, the OBEX client acknowledges the response packets by sending further Get request packets, until the OBEX server sends a final response packet with the Success response code, and prepares itself for the first request packet of the next command from the OBEX client. Upon receiving the response packet, the OBEX client can complete the object transfer, for example, commit the received data to file.

SetPath

The SetPath command is used by the OBEX client application to navigate the directory structure on the OBEX server. This directory structure will often map directly onto a real file system, but can also be used for virtual directory structures that describe different areas of a given service, for example, IrMC or the Bluetooth Phonebook Access Profile.

A SetPath command will always consist of a single request–response exchange. The OBEX client application determines the folder to move into, and encodes this into the request packet, along with flags which indicate whether to ‘back up’ a level first, and whether to create a named folder if it doesn’t currently exist. Upon receiving the request packet, the OBEX server will attempt to move to (and, if requested, create) the specified folder, before sending back a response packet with a response code indicating the success or failure of the SetPath command.

Note: the OBEX specification also indicates a defined XML format for folder listing objects that can be used on request to pass information on the folder structure of an OBEX service to the OBEX client. This format has been used in the OBEX FTP server example code.

Abort

The Abort command is used by the OBEX client application to abandon an ongoing command without taking the drastic step of disconnecting the OBEX transport. It is treated as a special case in the OBEX protocol, as an Abort request packet can be sent by the OBEX client instead of a

request packet that continues an ongoing OBEX command. Upon receipt of an Abort request packet, the OBEX server will abandon any ongoing activities associated with the current command. It will then send back an Abort response packet, and prepare itself for the first request packet of the next command from the OBEX client.

10.1.5 OBEX Transports

OBEX packets need a means to move between the OBEX client and the server. The means by which the packets are moved is known as the OBEX transport, and a particular instance of a connection established over an OBEX transport is known as an OBEX transport link.

OBEX can be run over many transports. The fundamental requirement of an OBEX transport is that it should be reliable – that is to say, transmission errors will be detected and handled in the transport layer. The OBEX protocol does not have any inbuilt mechanism to recover from malformed, missing or out of order packets. The transport responsible for sending the OBEX packets must ensure that they are delivered correctly and in the order of sending, or where an unrecoverable situation arises, notify OBEX that the transport link is no longer usable.

Bringing up an OBEX transport link

Bringing up an OBEX transport link is usually the responsibility of the OBEX client. In order to do this, the client will need some way to discover and address the OBEX server that it wishes to connect to. The process by which the client and server arrange for this to happen is called service discovery.

Service discovery

An OBEX server will generally use transport-specific mechanisms to advertise itself in order that OBEX clients are able to find it, work out what services it offers and how it can be addressed. Once it has created this advertisement, the OBEX server waits passively to be connected to.

Correspondingly, the OBEX client will use transport-specific mechanisms to discover the OBEX server (and the device on which it is running), and how to address the OBEX server. This information can then be used to establish the OBEX transport link so that packet exchange can begin.

Let's have a look at the transport-specific mechanisms.

Bluetooth service discovery OBEX servers running over Bluetooth use SDP to advertise services. Each Bluetooth service, OBEX or otherwise, has a corresponding SDP record which identifies the service being offered,

and the means to address that service in a Bluetooth-specific way. The address of an OBEX service running over Bluetooth is a combination of the RFCOMM server channel number and the Bluetooth device address. There may be additional requirements on an OBEX service to ensure that OBEX clients can connect to it over Bluetooth, such as ensuring that the device is connectable and, if dealing with an unpaired Bluetooth device, discoverable. The Bluetooth class of device (CoD), specifically the major service class, may also require modification.

OBEX clients running over Bluetooth can use the device discovery procedures identified in the GAP profile to identify devices offering object transfer services (based on the major service class bit field in the CoD) and their device addresses. Once the device has been found, the OBEX client can perform a more detailed query, using SDP, to gather details on the remote device's OBEX services, if any. The SDP query should indicate the type of OBEX services on offer, and provide the final necessary piece of addressing information, the RFCOMM server channel number.

The subject of SDP, along with discovering, connecting to and querying services (including getting and setting CoD information) on Bluetooth devices is covered in greater detail in Chapter 4.

IrDA service discovery OBEX servers running over IrDA use IAS to advertise services. The OBEX specification defines a well-known service, the 'Default OBEX server', identified by the IAS class name 'OBEX'. Other OBEX services must define their own specific class name with which the associated OBEX server will be registered and OBEX clients seeking that service can use to query IAS. An attribute 'IrDA:TinyTP:LsapSel' must be present for each registered OBEX class advertised using IAS, whose attribute value identifies the TinyTP LSAP to which the service is bound.

When a device offers an OBEX service over IrDA, the device must also set the OBEX hint bit in the IrLMP hint bitfield. This bitfield fulfils a similar purpose to the major service class bitfield in the CoD in Bluetooth.

More details on connecting to and querying services on IrDA devices can be found in Chapter 5.

Bringing down an OBEX transport link

An OBEX transport link can come down at any time, and any robust OBEX application must be prepared to deal with this. The transport link may come down because one of the OBEX peers decided that they would not or could not continue with the OBEX session, or because of link failure – Bluetooth devices can move out of range, IrDA beams can be blocked, etc.

10.2 OBEX in Symbian OS

10.2.1 Origins and Development

OBEX first appeared in EPOC Release 5 (ER5), the precursor to Symbian OS that was developed for the Psion Series 5mx. In that release, OBEX ran over the IrDA transport only and was intended for simple point-to-point beaming of items such as vCards, replacing an earlier Psion proprietary scheme.

The OBEX implementation in Symbian OS v9.1 is the result of incremental development of that first OBEX implementation, and the APIs are still to a large extent source compatible with that original version. The requirement to maintain source compatibility has resulted in some newer features being added in perhaps unexpected ways, as we will see later on.

10.2.2 Feature Set

The Symbian OS v9.1 OBEX implementation supports the following features:

- OBEX client and server (OBEX version 1.2)
- IrDA, Bluetooth and USB transports⁴
- OBEX authentication on connection
- All standard OBEX version 1.2 commands (Connect, Disconnect, Put, Get, SetPath, Abort)
- All standard OBEX headers and user-defined headers
- Transfer of OBEX objects to and from RAM or file
- Single or double buffering of received file data with user definable buffer sizes
- User-defined OBEX packet sizes
- Target/Who/Connection ID header support.

There are also some notable omissions from the Symbian OS OBEX implementation that an application developer should be aware of:

- Capability object (although this may be provided by higher level software)
- OBEX level routing/multiplexing.

⁴ USB resources, including the OBEX USB transport, are generally reserved for use by Symbian's licensees, so are not covered in this chapter.

Symbian OS v9.2 sees the continued development of the OBEX implementation, including:

- Improved OBEX server API to enable asynchronous processing of requests
- Ability to set timeouts for OBEX client commands

It is anticipated that future versions will also see the introduction of APIs for data streaming over OBEX, so that transfers are not limited by available RAM.

10.2.3 API Structure, Executable Structure and Header Files

The Symbian OS OBEX API splits into three parts: an API used to create objects for sending or receiving into; the OBEX client API; and the OBEX server API. In a typical application, the object API is used in conjunction with one of the other two APIs.

Applications can access these APIs by linking to `iobex.dll`. In this DLL, OBEX support is implemented as a group of interrelated classes whose code runs in the same thread as the application using OBEX. It is important to note that these classes include CActive-derived active objects that need their host thread to have an installed active scheduler to operate. This use of active objects within the OBEX implementation also has implications particularly for OBEX client application writers, who should avoid blocking the thread waiting for asynchronous OBEX commands to complete – in particular any use of `User::WaitForRequest()` or `WaitForAnyRequest()` will cause problems.

The headers that an application needs to include depend on the nature of the application, but in general following headers are used:

<code>obexobjects.h</code>	Included by OBEX applications which need to use the classes representing the OBEX object types (<code>CObexBaseObject</code> , <code>CObexBufObject</code> , <code>CObexFileObject</code> etc.). Effectively, this means all (useful!) OBEX applications
<code>obexheaders.h</code>	Included by OBEX applications which need to use classes representing OBEX headers and header sets – this means all OBEX applications using the recommended means to manipulate headers associated with OBEX object types
<code>obexclient.h</code>	Included by OBEX client applications

`obexserver.h` Included by OBEX server applications
`obexconstants.h` Included for access to classes defining (amongst other things) transport settings for OBEX clients and servers

Each of these API areas will now be covered in depth.

10.2.4 OBEX Object APIs

The OBEX object API provides a group of classes to represent OBEX objects (see Figure 10.4). The same classes can be used for both sending and receiving objects.

Each OBEX object consists of a group of (non-body) headers, and some representation of the body data. How the body data is represented varies depending on the class used. One of the two main OBEX object classes (`COBexFileObject`) allows body data to be sent from, and received to, a file; the other class (`COBexBufObject`) allows body data to be sent from, and received to, a RAM-based buffer. In fact, `COBexBufObject` also offers the ability to read from and write to files, as will be explored later.

COBexBaseObject

The representation of headers (other than the body headers) remains the same for all classes, and so this functionality is located in a base class, `COBexBaseObject`. This base class actually offers two distinct ways to handle non-body data.

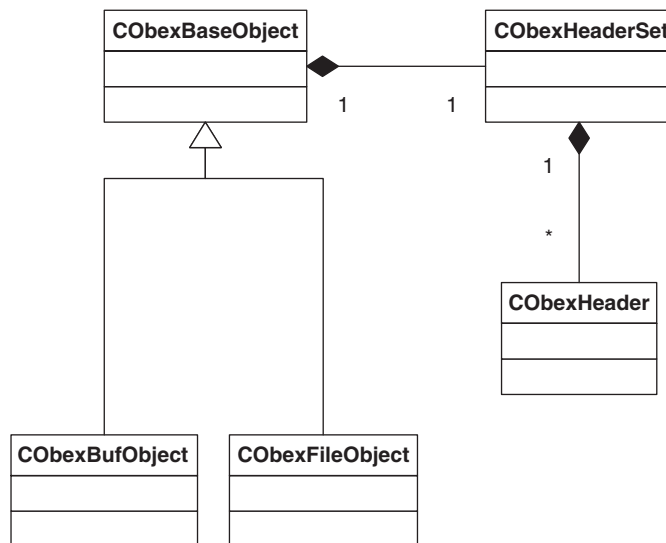


Figure 10.4 Symbian OS OBEX object classes

Old-style API

The original OBEX APIs offer a number of setter and getter function pairs, each pair being specific to a particular header, for example:

```
void CObexBaseObject::SetNameL(const TDesC& aDesc);
const TDesC& CObexBaseObject::Name();
```

When `SetNameL()` is called, the object would be updated to add a Name header. The presence or absence of a particular header in an object can be checked by using the `ValidHeaders()` function, which returns a `TObexHeaderMask`. `TObexHeaderMask` is a bit field made by using bitwise OR to combine values representing individual header types, such as `KObexHdrName` and `KObexHdrType`. These values are defined in `obexconstants.h`.

In normal use, `ValidHeaders()` is more useful when inspecting a received object, as the same bits are set when a header of a particular type is picked up during reception of an object. It can be used to decide if it is worth using the corresponding getter function to extract a given header from the object. Upon object creation or reset, the valid header bit field defaults to `0x0000`, indicating that no headers are valid (i.e., present in the object).

Once an object has had all the appropriate headers set, an attempt to transfer that object using a Put or a Get command will normally send all the valid headers. The sending of any particular header can be suppressed by resetting (i.e., making equal to 0) the corresponding bit of the *header mask*. The header mask is another bit field using the same values defined in `obexconstants.h`, but in this case, the bit corresponding to a particular header type is set to 0 if the header is not to be sent, or 1 if it should be sent. This header mask bit field defaults to `0xFFFF`, indicating that all the valid headers for the object should be sent. The application writer may use the getter and setter function pair for the header mask to extract and modify this default value. (In practice, it is not clear how useful this feature actually is!)

The remaining peculiarity of the old-style API is the handling of HTTP headers. HTTP headers are slightly unusual in that there can be more than one in a given object. This leads to slightly unusual semantics to the HTTP setter and getter functions:

```
void CObexBaseObject::AddHttpL(const TDesC8& aDesc);
const RPointerArray<HBufC8>* CObexBaseObject::Http() const;
```

Please note the ‘add’ semantics of the setter function, and the array return value of the getter function. Unlike the other setter functions, the `AddHttpL()` function simply appends another HTTP header to the set belonging to the object.

New-style API

The new-style API was introduced in Symbian OS v7.0s and has been available since then.

While the old-style API sufficed for a number of releases, it was not particularly scalable. When support for user-defined header values was introduced into the OBEX implementation, it was necessary to rethink the header handling in the API. A more scalable and consistent means was devised to accommodate the setting and getting of headers.

While the individual setter and getter pairs remained for backward source and binary compatibility, three new functions were added to `COBexBaseObject` to allow header manipulation:

```
void COBexBaseObject::AddHeaderL(COBexHeader& aHeader);
const COBexHeaderSet& COBexBaseObject::HeaderSet() const;
COBexHeaderSet& COBexBaseObject::HeaderSet();
```

These new functions model the representation of headers within the object; each `COBexBaseObject` (or `COBexBaseObject` derived class) has a *header set*, made up of an arbitrary collection of headers. Two new classes were introduced to directly represent these to the application writer: `COBexHeaderSet` and `COBexHeader`. Each `COBexBaseObject` owns a `COBexHeaderSet`, which in turn will own zero or more `COBexHeader` objects. The functions listed above operate on the `COBexHeaderSet` owned by the `COBexBaseObject` and allow the set to be added to and accessed in a `const` form and non-`const` form, respectively.

For an object created for transfer to another device, the `COBexHeaderSet` belonging to the `COBexBaseObject` defines the set of headers that will be sent during a transfer (except Body and End-of-Body headers). For a received object, the `COBexHeaderSet` defines the set of headers received during the transfer. The `COBexHeaderSet` can be iterated through using an `MOBexHeaderCheck`-derived class (of which more later).

COBexHeader, COBexHeaderSet and MOBexHeaderCheck

The application writer should note a couple of things about these classes. Firstly, they are both Symbian C-style classes, meaning that they are derived from the `CBase` class and intended for use solely on the heap.

Secondly, an instance of `COBexHeader` is actually a handle to an ‘underlying’ header representation, as shown in Figure 10.5. This allows multiple `COBexHeader` instances to share a single underlying header

representation, thereby saving RAM (and potentially a lot of RAM, depending on how big the header representation is, and how many copies of it are required within an application). The application writer needs to be aware of the implications of this.

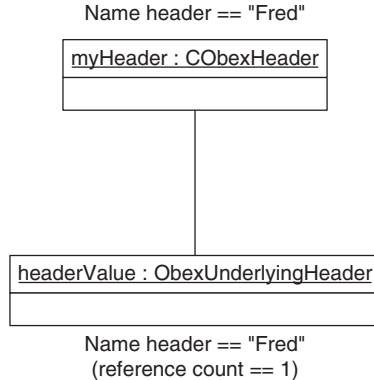


Figure 10.5 `CObexHeader` internal representation

When a copy of one instance of `CObexHeader` is created using the `CObexHeader::CopyL()` function, the new copy will share the same underlying header representation, that is, it is a shallow copy, as shown in Figure 10.6.

Also, when one instance of `CObexHeader` is set to the value of another instance of `CObexHeader` using the `CObexHeader::Set()` function, both instances will share the same underlying header representation.

Most of the time, this sharing of underlying headers is transparent to the application writer; a reference count of the number of `CObexHeader` instances referencing the underlying header representation is kept automatically. An underlying header will only be deleted when the last `CObexHeader` referencing is deleted.

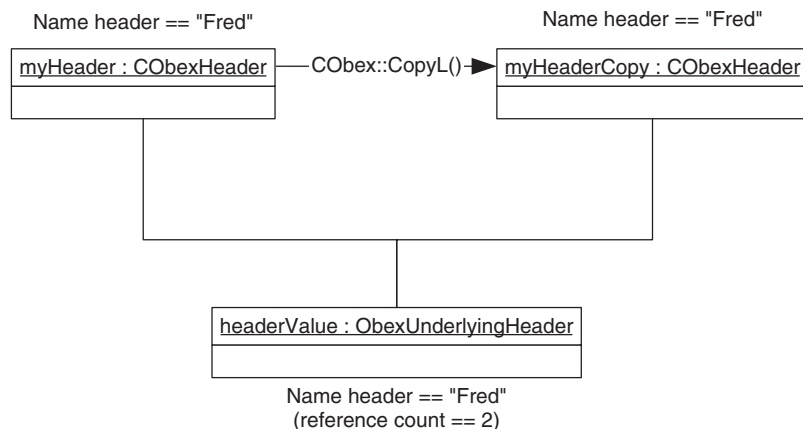


Figure 10.6 `CObexHeaders` with shared internal representation

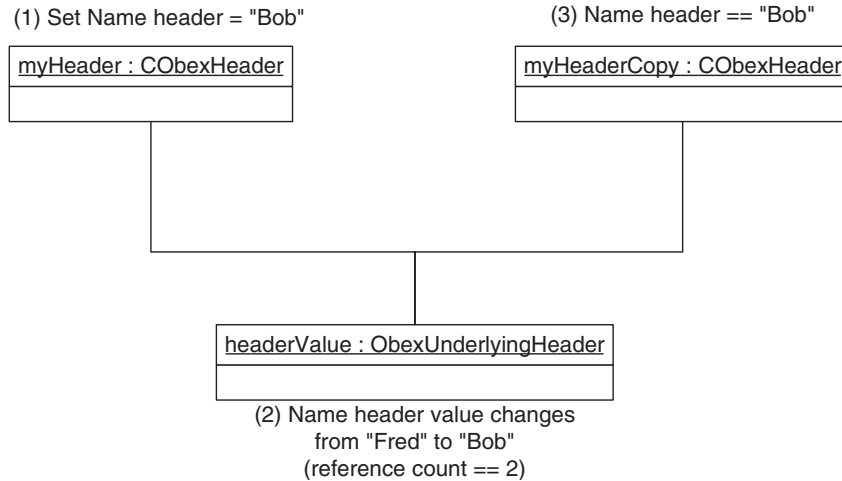


Figure 10.7 Consequence of shared internal representation

Application writers need to be particularly aware of the sharing of underlying headers when the header value is changed. In this case, all COBexHeader instances that reference the changed underlying header will effectively have their values updated, as shown in Figure 10.7.

Using COBexHeader If you haven't already, now is a good time to take a look at `obexheaders.h` to get a feel for COBexHeader. Its main features are a static factory function (`NewL()`) used to create a new instance, a group of setter functions corresponding to each of the four encoding types, and a group of getter functions again corresponding to each of the four encoding types. There is also a function (`Type()`) which can be used to determine which encoding type has been used for a given header.

Below is an example of setting and inspecting a COBexHeader.

```
// Some constant definitions
const TUint8 KTypeHeaderHi = 0x42;
_LIT8(KFolderListingType, "x-obex/folder-listing\x00");
_LIT(KOutputFormat, "Byte Sequence Value = %S")
_LIT(KFolderListingOutput, "Type header, value == folder listing\n");

// Create new instance of COBexHeader on the heap
COBexHeader* header = COBexHeader::NewL(); // Has no value yet
CleanupStack::PushL(header); // Place on cleanup stack
header->SetByteSeqL(KTypeHeaderHi,
                   KFolderListingType); // Set value
CleanupStack::Pop(header); // Remove from cleanup stack

// {Intervening code that might change value of header}
// Check that value of header is still the same...
//
if ( header->HI() == KTypeHeaderHi &&
    header->AsByteSeq() == KFolderListingType )
```

```

{
    // Value of header is unchanged
}
delete header;    // destroy header

```

Example 10.1 Use of CObexHeader

Example 10.1 shows the creation of a CObexHeader instance. Initially, the CObexHeader has no value. The value is then set using SetByteSeqL().

There are two things of interest here; firstly, we use SetByteSeqL() since the HI indicates a byte sequence encoding (HI == 0x42, so two high order bits == 01 in binary). Also note that because SetByteSeqL() has to allocate memory to store a copy of the folder listing type string it is a leaving function, denoted by the trailing 'L' in the function name. This is the reason that the header is placed on the cleanup stack before being set. In this respect, SetByteSeqL() is similar to SetUnicodeL(), but dissimilar to SetByte() and SetFourByte() which do not allocate any memory and therefore do not leave.

The next part of the code illustrates how a header can be inspected. The HI of the header is checked first. This effectively checks two things. Firstly, it ensures that the right header has been found (e.g., the Type header rather than the Name header). Secondly, because the encoding type is held as part of the HI, it ensures that the header is using the correct encoding type. This is an important step, since attempting to extract the header value using an inappropriate getter function will result in a panic.

Once it is established that the HI is correct, AsByteSeq() is used to extract the value of the header so that a comparison can be made with the required value. The header is then finally deleted. Unless another CObexHeader has been created which references the same underlying header value, this underlying header value will also be deleted.

Using CObexHeaderSet Setting and inspecting individual headers is useful, but most often headers are dealt with in groups. CObexHeaderSet was created as a container class for groups of headers.

Its main features are a static factory function (NewL()) to create a new instance of CObexHeaderSet, a means to add and delete headers, a means to find specific headers, and finally a means to iterate through the headers in the collection. As a result of the iteration feature, each CObexHeaderSet has a concept of the 'current element', which can be reset to the first element of the set, or moved on to the next element.

Below is an example of how to use some of these features.

```

// Start with some constant definitions
//
const TUint8 KNameHeaderHi = 0x01;

```

```

const TUint8 KLengthHeaderHi = 0xC3;
const TUint8 KUserDefinedHi = 0x70;

_LIT(KObjectName, "This is the name of the object");
const TInt KObjectLength = 400;
_LIT(KUserDefinedByteSeqValue, "This could be anything, but must be zero
    terminated\x00");

// Create header set
//
CObexHeaderSet* headerSet = CObexHeaderSet::NewL();
CleanupStack::PushL(headerSet);

// Create and add first header
//
CObexHeader* header1 = CObexHeader::NewL();
CleanupStack::PushL(header1);
header1->SetUnicodeL(KNameHeaderHi, KObjectName);    // Set value
User::LeaveIfError(headerSet->AddHeader(header1));
CleanupStack::Pop(header1);    // Ownership has been passed to
                                // CObexHeaderSet

// Create and add second header
//
CObexHeader* header2 = CObexHeader::NewL();
header2->SetFourByte(KLengthHeaderHi, KObjectLength); // Set value
CleanupStack::PushL(header2);
User::LeaveIfError(headerSet->AddHeader(header2));
CleanupStack::Pop(header2);    // Ownership has been passed to
                                // CObexHeaderSet

// Create and add third header
//
CObexHeader* header3 = CObexHeader::NewL();
CleanupStack::PushL(header3);
header3->SetByteSeqL(KUserDefinedHi, KUserDefinedByteSeqValue);
User::LeaveIfError(headerSet->AddHeader(header3));
CleanupStack::Pop(header3);    //Ownership has been passed to
                                // CObexHeaderSet

```

Example 10.2 Creating and populating a CObexHeaderSet

Example 10.2 shows the creation and population of a CObexHeaderSet object. Note particularly the use of the cleanup stack, as this indicates how ownership of the header (i.e., the responsibility for deleting the header) passes to the CObexHeaderSet when AddHeader() is called. Once AddHeader() has returned successfully, the CObexHeaderSet has responsibility for eventually deleting the header that was passed in, and so whatever class or function had responsibility for this before can relinquish that responsibility (in this case, the calling function initially has ownership, but gives that up following AddHeader() by using CleanupStack::Pop()).

```

_LIT(KLengthOutputString, "Length == %d\n");

```

```
COBexHeader* findHeader = COBexHeader::NewL();

headerSet->First();
if ( headerSet->Find(KLengthHeaderHi,*findHeader)==KErrNone )
{
    iTest.Printf(KLengthOutputString, findHeader->AsFourByte());
}

delete findHeader;
```

Example 10.3 Finding a header in a header set

Example 10.3 shows how a populated header set can be easily searched for a specific header. Note that `Find()` searches from the current element of the `COBexHeaderSet` onwards. This is the reason for the use of `First()`, to reset the current element to be the start of the set.

(Now is a good time for a quick diversion to discuss ordering within header sets. Elements – that is, headers – in a `COBexHeaderSet` are ordered but not sorted; they are simply stored in order of addition. The element that was added earliest is at the start of the set, and the one added most recently at the end of the set. Elements can be deleted from anywhere in the set, but additions always happen at the end.)

Once the current element has been reset to the start of the set, `Find()` can be used to seek the first instance of a header with the specified HI. If such a header is found, the function will return `KErrNone` to indicate success, and the `COBexHeader` passed in by reference will be updated to the value of the matching header. If no such header is found, the function will return an error code and the header parameter will not be changed.

```
_LIT(KUnicodeOutputString, "Unicode, HI = 0x%02x, Value = %S\n");
_LIT(KByteSeqOutputString, "Byte Seq, HI = 0x%02x, Value = %S\n");
_LIT(KOneByteOutputString, "One Byte, HI = 0x%02x, Value = %d\n");
_LIT(KFourByteOutputString, "Four Byte, HI = 0x%02x, Value = %d\n");

COBexHeader* iterHeader = COBexHeader::NewL();
CleanupStack::PushL(iterHeader);

headerSet->First();
while(headerSet->This(iterHeader) == KErrNone)
{
    switch (iterHeader->Type())
    {
        case COBexHeader::EUnicode:
            iTest.Printf(KUnicodeOutputString,
                iterHeader->HI(),
                &(iterHeader->AsUnicode()));
            break;

        case COBexHeader::EByteSeq:
            {
                HBufC* buf=HBufC::NewL(iterHeader->AsByteSeq().Length());
```

```

buf->Des().Copy(iterHeader->AsByteSeq());
iTest.Printf(KByteSeqOutputString, iterHeader->HI(), buf);
delete buf;
buf=NULL;
}
break;

case CObexHeader::EByte:
    iTest.Printf(KOneByteOutputString,
        iterHeader->HI(),
        iterHeader->AsByte());
    break;

case CObexHeader::EFourByte:
    iTest.Printf(KFourByteOutputString,
        iterHeader->HI(),
        iterHeader->AsFourByte());
    break;

default:
    break;
}
headerSet->Next();
}

CleanupStack::PopAndDestroy(iterHeader);

```

Example 10.4 Iteration through a header set

Example 10.4 shows iteration through a header set. Again, `First()` is used to reset the current element to the start of the header set before `This()` is used to attempt to get the value of the current element.

If there is a current element, `This()` will return `KErrNone`, and will update the `CObexHeader` passed in by reference to be the value of the current header. If there are no elements in the set, or iteration has moved the current element beyond the end of the set, `This()` will return an error code.

Using `MObexHeaderCheck` In some circumstances, it can be useful to apply a filter to a header set, so that only headers of interest are included in a search or an iteration through a header set. The means to achieve this is by the use of an `MObexHeaderCheck`-derived class.

`MObexHeaderCheck` is an example of a ‘mixin’ class: a class with only virtual (and often just pure virtual) functions and no member data. They are used to define an interface that can be inherited by any class that is willing to implement the behaviour of all the functions.

`MObexHeaderCheck` has a very simple API; it defines a single function⁵ that must be implemented by concrete derived classes, namely

⁵ Actually, in addition to this `MObexHeaderCheck` also has a `Reset()` function and an extension interface function, both of which have default implementations (i.e., they are virtual rather than pure virtual). Neither of these functions is normally of interest to the application writer.

`Interested()`. As a parameter, `Interested()` takes a `TUInt8` representing the HI value of a header, and returns a `TBool`. The developer who implements `Interested` must ensure that the `TBool` return value is set to `ETrue` if headers with this HI are of interest, and to `EFalse` if they are not of interest. This decision forms the basis of the filter. An example of such an implementation follows.

```
class TNameAndLengthMask : public MOBexHeaderCheck
{
public:
    virtual TBool Interested(TUInt8 aHi)
    { return aHi == KNameHeaderHi || aHi == KLengthHeaderHi; };
};
```

Example 10.5 Example of `MOBexHeaderCheck` implementation

Classes like the one defined above could be used in a number of different ways:

- It can be used as a mask during iteration. By passing in the address of an instance of such a class to `COBexHeaderSet::SetMask()` prior to beginning an iteration, only headers representing the name header and length header would appear to be in the header set.
- By passing in the address of an instance of such a class to `COBexHeaderSet::SetMask()`, it can be used to delete the headers that are not of interest from the header set, by using `COBexHeaderSet::DeleteMasked()`.
- It can be used to make a copy of the header set that includes only the interesting headers, by using `COBexHeaderSet::CopyL()` and passing in an instance of such a class.

Having dealt with how non-body headers are handled in the `COBexBaseObject`, we now move onto seeing how body headers are handled in `COBexBaseObject`'s derived classes. We start with the most important of these, `COBexBufObject`.

COBexBufObject

`COBexBufObject` was originally intended as a means to send body data from, and receive body data to, a RAM-based buffer. Over time, it has been extended and modified beyond this original remit and now allows receiving to files using a number of buffering strategies, and also sending from files. These different strategies can be used to increase the performance of OBEX transfers, and will be discussed individually below.

In general, the way `COBexBufObject` works when receiving an object depends primarily on whether it is being used for sending (as a source) or receiving (as a sink), and what combination of file and buffer it has been set up with. The combinations are listed in Table 10.4.

On construction using the `NewL()` factory function, the `COBexBufObject` takes a pointer to a `CBufFlat`. This was the original method of setting up a `COBexBufObject` for use, and without further modification, the object will behave as in the 'Buffer only' column. Ownership of the buffer remains with the code creating the `COBexBufObject`, so you should be careful to explicitly delete the buffer when finished.

It should also be noted that the pointer value passed in may be `NULL`, as the OBEX application can decide to set up the `COBexBufObject` for transfer after construction time.

To set up a `COBexBufObject` to a different file and buffer combination, the OBEX application writer has a few choices – this is again a combination of the gradual expansion of the capabilities of the `COBexBufObject`, and the necessity of maintaining source and binary compatibility.⁶ The choices are:

Table 10.4 OBEX behaviour with differing buffer/file combinations

File/buffer setup Usage	RAM based buffer only (using <code>TOBexBuffering</code> Details)	File only (using <code>TOBexPureFileBuffer</code>)	RAM-based buffer and file (using <code>TOBexFilenameBackedBuffer</code> or <code>TOBexRFileBackedBuffer</code>)
For sending	Body data is extracted from buffer	Body data is read directly from file	Body data is read directly from file
For receiving	Body data is received to buffer, with buffer being expanded as necessary to accommodate new body data	Body data is written directly to file, with no buffering	Body data is received to buffer. When buffer is filled (or transfer is complete) the buffer is written to file Further modifications to this buffering strategy can be made in this combination – see below

⁶ Actually, the version of OBEX in version 7.0 of Symbian OS was not entirely source and binary compatible with other pre-v9.1 releases due primarily to differences in the `MOBexServerNotify` class. There has also been a binary (but not source) compatibility break between the pre-v9.1 releases and v9.1, to accommodate a new compiler and platform security. From v9.1 onwards, source and binary compatibility will be maintained for future releases.

- Old-style API: use one of the various overloads of `COBexBufObject::SetDataBufL()` that take combinations of `CBufFlat` and descriptors containing filenames.
- New-style API: use the overload of `COBexBufObject::SetDataBufL()` that takes a reference to a `TOBexBufferingDetails`.

The new-style API and the `TOBexBufferingDetails` class were introduced to stop the `COBexBufObject` API growing every time a new variation of its usage was devised. This is the recommended means to set up the `COBexBufObject` ready for use.

The `TOBexBufferingDetails` class only contains the basic information required for setting up `COBexBufObject` with a buffer, but has a number of derived classes, which provide additional information for setting up the `COBexBufObject` for use with a combination of buffer and file.

There is a further (important) subtlety to be aware of when using a `CBufFlat` for receiving data; when initially created, a `CBufFlat` is of zero size. This is fine for use in conjunction with `COBexBufObject` in a 'buffer only' setup, because the buffer is automatically expanded to accept new data. However, when a `CBufFlat` of zero size is used in conjunction with `COBexBufObject` in a 'buffer and file' setup, the buffer is *not* automatically expanded. For this reason, it is important to use `CBufFlat::ResizeL()` to adjust the size of the buffer before beginning a transfer in this case.

This, and the use of `SetDataBufL()`, is illustrated by the following sequence of code fragments from the example OBEX File Transfer Profile application.

```
// Create a buffer to be used when receiving data
//
iReceiveBuf = CBufFlat::NewL(KExpandSize);

// Important to remember that when a buffer is being
// used as a temporary store for received data before the
// data is ultimately committed to file, OBEX will not
// expand the buffer as data arrives. Therefore, the
// application must size the buffer as well as create it
// before use.
//
iReceiveBuf->ResizeL(KBufSize);

// TOBexFilenameBackedBuffer allows the application writer
// to specify the buffer size, filename and buffering
// strategy to be used during reception.
//
TOBexFilenameBackedBuffer bufDetails(*iReceiveBuf,
    iTempFileName, COBexBufObject::ESingleBuffering);

// For the moment, we provide COBexBufObject with a NULL
// buffer pointer.
//
```

```
iReceiveObject = COBexBufObject::NewL(NULL);

// Call SetDataBufL to prepare iReceiveObject for transfer.
//
iReceiveObject->SetDataBufL(bufDetails);
```

Example 10.6 Creating a COBexBufObject for reception

TOBexBufferingDetails buffering strategy TOBexBufferingDetails contains the details of a CBufFlat, and is used for sending and receiving from a RAM-based buffer. This makes it ideal for sending small objects that can be constructed in RAM buffers, or receiving objects that can be extracted easily from RAM buffers. As mentioned above, when being used for data reception, the CBufFlat will be expanded to accommodate the incoming body data.

TOBexPureFileBuffer buffering strategy TOBexPureFileBuffer is derived from TOBexBufferingDetails, and adds a file name to the information used to set up the COBexBufObject. The file name is the name of the file that body data should be sent from or received to when the COBexBufObject is used in an OBEX transfer.

Immediately on being provided with TOBexPureFileBuffer, OBEX will attempt to open the named file for read/write access, or, failing that, read only access. If the file does not exist, OBEX will attempt to create a file with the specified name. If none of these steps succeed, an error will result. When receiving to the file, any pre-existing file contents are erased before reception commences.

This class provides a special case where no buffering occurs when receiving body data. For performance purposes, it is generally advisable to use one of TOBexFilenameBackedBuffer or TOBexRFileBackedBuffer in preference to this class.

TOBexFilenameBackedBuffer buffering strategy TOBexFilenameBackedBuffer is derived from TOBexBufferingDetails, and again adds a file name to the information used to set up the COBexBufObject.

The behaviour of this class is identical to that of TOBexPureFileBuffer in all but one important aspect. Unlike TOBexPureFileBuffer, this class provides buffering when receiving data, and is the class most commonly used by OBEX applications.

COBexBufObject's receive buffering works by accumulating body data from successive OBEX packets into a RAM buffer, and only writing the accumulated data to file when the buffer is full, or the transfer is complete. This improves transfer performance because making a few large file writes is much more efficient than making a large number of small file writes.

When choosing the size of the buffer, the application writer should take into consideration the likely amount of body data being received within a

particular transfer, the amount of body data that might be transferred in an OBEX packet (see the section ‘Maximum OBEX packet sizes’ on varying OBEX packet sizes), and the likely availability of heap space. Ideally, the application should create a buffer that is big enough to contain an entire transfer’s worth of body data, but the penalty for adopting this approach is increased RAM usage. If the amount of body data is too large to make that practical or desirable, to make the buffer worthwhile it must at least be large enough to accommodate multiple OBEX packets’ worth of body data to see any performance gain. However, these are rough heuristics, and some experimentation might be necessary to find the ideal RAM/performance trade off for a given application.

TObexRFileBackedBuffer buffering strategy TObexRFileBackedBuffer is derived from TObexBufferingDetails, and is almost identical in TObexFilenameBackedBuffer. The difference is that rather than adding a filename to the base class, it adds a file handle in the form of an RFile object. This, in conjunction with the handle-sharing scheme in Symbian OS, can allow access to a file in another (cooperating) application’s private directory.

The RFile should have already been opened when it is passed into the TObexRFileBackedBuffer constructor. OBEX will use the file referenced by this handle in the same way that it would a file named in a TObexFilenameBackedBuffer structure.

CObexFileObject

The CObexFileObject provides a means to send body data from, and receive body data to, a file. For sending, it performs the same task as a CObexBufObject configured with a file (that is, with a TObexPureFileBuffer, TObexFilenameBackedBuffer, or TObexRFileBackedBuffer object). For receiving, it performs the same task as a CObexBufObject configured with a filename only (that is, with a TObexPureFileBuffer). For the best performance when receiving body data to a file, it is advisable to use CObexBufObject configured with a file and buffer in preference to CObexFileObject.

CObexNullObject

The CObexNullObject is used where there is no body data to send, or where the receiving code is only concerned with non-body headers. Although used internally in the OBEX implementation, this class is of limited use in general applications.

10.2.5 OBEX Client API

The OBEX client API in Symbian OS is defined by the class CObexClient. An object of this class must be instantiated by any application

wishing to issue OBEX commands to remote devices.⁷ The part of the API used for initiating OBEX commands is asynchronous, and uses the standard mechanism built around using a `TRequestStatus` to signal completion of an OBEX command. This allows OBEX commands to be run (effectively) in parallel with other application activities using the active object framework.

Important note: the OBEX implementation is built of various active objects, and so to process OBEX commands, it relies upon the active scheduler not being blocked. This means that OBEX applications that make asynchronous requests to `COBexClient` must not use `User::WaitForRequest()` or any other mechanism that will block the thread to wait for completion of the request. This is a 100 % guaranteed way to terminally hang the thread in which the `COBexClient` is running! The usual way to handle such completions is to have an active object await completion of the request, and to take appropriate action in its `RunL()` function.

This parallelism does not extend to other OBEX commands though; each instance of `COBexClient` may have only one OBEX command request in progress at one time.⁸ An attempt by an OBEX client application to initiate another OBEX command when one is already in progress will result in the new OBEX command completing immediately with `KErrAccessDenied`.

An OBEX client application instantiates its `COBexClient` using the `COBexClient::NewL()` factory function. At the time of creation, the application must have made choices on the transport over which OBEX should run, and (maybe) the intended target device and service.

Creation of `COBexClient` and selection of transport

There are actually two overloads of `COBexClient::NewL()` in Symbian OS v9.1, and three in v9.2.

The original version takes a reference to a `TOBexProtocolInfo` structure. This is a base class that does nothing more than identify the transport type to be used (IrDA or Bluetooth), and is not of any use by itself. It does, however, have a number of subclasses that define transport specific details for each transport type, and it is instances of these

⁷ In fact, an application writer does have other options if they want to simply send an object from one device to another – this is covered in Chapter 9, with the description of the `SendAs` API. Some of the transports offered through the `SendAs` API use OBEX themselves, but hide all of the details away from the API user.

⁸ *Abort* is an exception to this rule as it can be called while OBEX commands are ongoing. This is because `Abort` is used to bring ongoing OBEX commands to an end, and so has to be available at all times.

subclasses that are passed into `COBexClient::NewL()` by applications using OBEX.

Before introducing the subclasses of `TOBexProtocolInfo`, it is worth considering the characteristics of the transport types and protocols, to understand the differences in the way the transport is set up.

OBEX client connections over IrDA

For OBEX client applications running over the IrDA infrared protocols, transport set up is trivial; the client application has no need to explicitly identify a specific device to be communicated with, because the line-of-sight characteristic of infrared means that the user can intuitively ‘point and shoot’ at the device on which the required OBEX server is running⁹. Additionally, service discovery on IrDA is relatively straightforward – given a class and service name, it is possible to use the IrDA Information Access Service (IAS) to determine the appropriate Tiny TP (another IrDA protocol) port number to connect to in order to access that service.¹⁰

As a result, an OBEX client application running over IrDA need only identify the class and name of the required service. When the client application later initiates an OBEX Connect, `COBexClient` will make an IrDA connection to the device in line of sight, and performs IAS queries to locate the required service on that device.

The subclass of `TOBexProtocolInfo` used to hold IrDA transport information is `TOBexIrProtocolInfo`. `TOBexIrProtocolInfo` adds a number of new fields to the `iTransport` field it inherits from `TOBexProtocolInfo`.

When a `TOBexIrProtocolInfo` object is created, its `iTransport` member must be set to the value of the literal constant `KOBexIrTTPProtocol`. This allows `COBexClient` to correctly interpret the rest of the fields in the class. (Where the optional `iDiscoverySlots` and `iDiscoveryAttempts` fields are set, `iTransport` must be set to the value of the constant `KOBexIrTTPProtocolV2`). The values of these constants are defined in `obexconstants.h`.

The first IrDA specific field in `TOBexIrProtocolInfo` is a `TIrdaSockAddr` (socket address) field. The `TIrdaSockAddr` class is defined by the infrared subsystem, and is itself made up a group of fields that combine to describe the two ends of an IrDA connection. This field is actually unused when creating a `COBexClient`.

The most important fields of `TOBexIrProtocolInfo` for an OBEX Client are `iClassName` and `iAttributeName`. These descriptor-based

⁹ Actually, IrDA is capable of more sophisticated point to multipoint modes of operation, but OBEX over IrDA is normally used as part of a point-and-shoot operation. Hence, when requested to connect over IrDA while in range of a number of suitable IrDA devices, the Symbian OBEX client will attempt to connect to the first one it successfully discovers.

¹⁰ As stated earlier, more details of IrDA-specific details can be found in Chapter 5.

fields are used to indicate the service to which the OBEX client wishes to connect.

```
TObexIrProtocolInfo info;
_LIT8(KObexClass, "OBEX");
_LIT8(KTinyTpLsapSel, "IrDA:TinyTP:LsapSel");

info.iTransport = KObexIrTTPProtocol;
info.iClassName = KObexClass;
info.iAttributeName = KTinyTpLsapSel;

iClient = COBexClient::NewL (info);
```

Example 10.7 Creating a **COBexClient** to run over IrDA

The remaining fields of `TObexIrProtocolInfo` are `iDiscoverySlots` and `iDiscoveryAttempts`, and are optional. These fields are used to modify the way in which the IrDA transport link is established. Note how the `iTransport` member in the following example is set to `KObexIrTTPProtocolV2` rather than `KObexIrTTPProtocol`. This distinction is used by OBEX to determine whether to access and use the value in `iDiscoverySlots` and `iDiscoveryAttempts`, or whether to use the default values for these two settings.

```
TObexIrProtocolInfo info;
_LIT8(KObexClass, "OBEX");
_LIT8(KTinyTpLsapSel, "IrDA:TinyTP:LsapSel");

info.iTransport = KObexIrTTPProtocolV2;
info.iClassName = KObexClass;
info.iAttributeName = KTinyTpLsapSel;
info.iDiscoverySlots = KNumberOfDiscoverySlots;
info.iDiscoveryAttempts = KNumberOfDiscoveryAttempts;

iClient = COBexClient::NewL(info);
```

Example 10.8 Creating a **COBexClient** to run over IrDA with modified IrDA connection behaviour

OBEX client connections over Bluetooth

For Bluetooth, the task is somewhat more involved. Bluetooth has no concept of line of sight so other means must be used to identify the device to which we are going to connect. For this reason, an OBEX client application must provide a Bluetooth device address to `COBexClient` to identify the device on which the required service is running. The normal means of identifying such a device in the first place is to use the device discovery procedures covered in Chapter 4.

Furthermore, the Bluetooth Service Discovery Protocol (SDP) is rather more complex than IAS, and in addition to basic information such as port numbers, a service record in the SDP database may contain information

which a client application needs to be aware of. For example, in the Bluetooth Object Push Profile (OPP), part of the SDP record is allocated to specifying the formats of objects that the OBEX server application implementing OPP can understand. This is definitely application, rather than OBEX, specific detail.

For this reason, interrogating the SDP database on a target device is left to the OBEX client application, and is beyond the scope of this chapter. SDP is covered in more detail in Chapter 4.

As a result of this, a client application using OBEX over Bluetooth must provide both a Bluetooth device address and a port number (specifically, an RFCOMM server channel number). The subclass of `TObexProtocolInfo` used to hold this information is `TObexBluetoothProtocolInfo`.

Maximum OBEX packet sizes

The second overload of `COBexClient::NewL()` also takes a reference to a (subclass of) `TObexProtocolInfo` object as a parameter, but additionally takes a reference to an object of class `TObexProtocolPolicy`. This class allows finer control over the maximum packet size used by OBEX, which in turn allows tuning of the performance of OBEX transfers.

The `TObexProtocolPolicy` offers two main functions, `SetReceiveMtu()`¹¹ and `SetTransmitMtu()`. These functions can be used to specify the size of buffer that is created by `COBexClient` to accommodate incoming packets (`SetReceiveMtu()`), and outgoing packets (`SetTransmitMtu()`).

In practice, there is no guarantee that the full size of the allocated buffers will be used in an OBEX transfer. This depends on the capabilities of the other device. The receiving capabilities of an OBEX client and server are determined during connection establishment, each side telling the other of the largest incoming packet it can accommodate. If the receiver can only accommodate packets smaller than those a sender is capable of sending, the sender ‘throttles back’ the size of the packets it sends so that they do not exceed the receiving buffer size of the receiving device.

It should be noted that this API offers the ability to independently set the receive MTU and the transmit MTU: this is useful where a given application of OBEX involves transfers predominantly in one direction. For example, where a given application involves an OBEX client sending large amounts of data to an OBEX server, the OBEX client’s transmit MTU might be increased to the maximum allowable, while the receive MTU might be left at the default value, or even reduced below that default.

¹¹ MTU = maximum transfer unit, a general term use to describe the size of the largest packet of data that may be exchanged by a given layer of a communications protocol.

Manipulating maximum OBEX packet sizes for improved performance

We will diverge from discussion of the OBEX client API for a moment to consider performance tuning and how this relates to OBEX packet size.

The default packet size used for OBEX in Symbian OS is 4000 bytes per packet (reflecting the original use of OBEX as a means to beam small objects like vCards around). The maximum allowable size for an OBEX packet as defined in the specification is 2^{16} (65,536) bytes. There is therefore quite some scope for increasing the maximum packet size.

Generally, the larger the size of an OBEX packet, the better the performance of the transfer. This is partly because the ratio of OBEX packet header information to OBEX packet payload is lower with a large packet. Mainly though, it is because more payload per packet means fewer packets are required to move an object of a given size, thus reducing the number of times that the processing overhead associated with receiving a packet is incurred. (Very generally, this overhead involves parsing the incoming packet, constructing and sending another packet in response, and then waiting for another packet.) This is particularly true with a conversational protocol like OBEX, where every packet sent from a client must be acknowledged before the next packet can be sent. This can be contrasted with Bluetooth's RFCOMM with its credit-based flow control, or TCP/IP's 'sliding window' flow control mechanism.

This 'bigger is better' maxim does not always hold true, however. When sending data, allocating a large maximum packet size is no use if the object to be transferred is smaller than the packet (where some of the allocated packet buffer space is unused), or if the receiving device is unable to handle packets that large. Equally, when receiving data, a large maximum packet size is no use if the device sending the data is only capable of sending packets smaller than the maximum size. In these cases, the unused capacity of the buffers used to store incoming and outgoing packets is simply wasted RAM.

The aim in setting the maximum packet sizes is therefore to balance improved performance against memory consumption. If RAM were not a consideration and OBEX were operating in isolation, the ideal maximum packet size would be the smaller of the 65,536 byte limit imposed by the OBEX protocol and the total size of the object to be transferred. However, RAM conservation should always be a consideration when implementing code for a Symbian OS device, so this may not be practical. Additionally, OBEX does not operate in isolation – the data transferred using OBEX is going to be processed in some way (e.g., music files might be written to disk, vCards transferred might be deposited in a database). This processing might introduce throughput limitations that mean there is no point in using additional RAM to improve OBEX throughput beyond a certain point.

The 4000 byte default maximum packet size in Symbian OS is at the low end of the available range – for applications that will be transferring larger objects, increasing this to a packet size of 32 kB should

give a reasonable performance without excessive memory consumption, while allowing reasonable responsiveness for Abort operations. As may be appreciated though, performance, especially where protocols is concerned, is a very complex subject, and experimentation is probably the best way to determine the optimum packet sizes for any given application.

Note: for interoperability purposes with some other IrDA implementations, the OBEX packet size for the IrDA transport has been fixed at the negotiated payload size of the underlying IrDA Tiny TP session.

ConnectL

Once an OBEX client application has created its `COBexClient` object, usually the next step is to attempt to initiate an OBEX session. The client application can achieve this by using `COBexClient::Connect()` or `ConnectL()`.

`COBexClient` offers a number of overloads of `Connect()` and `ConnectL()`, and the overload to use depends upon two things: whether the server is to be authenticated during connection, and whether there is data that must be transferred as part of the connection.

For situations where authentication of the server is not required and there is no data to be transferred during connection, there is a simple `Connect()` overload that takes a single `TRequestStatus&` parameter.

The `TRequestStatus` is used to signal completion upon success or failure of the connection attempt. When the `Connect()` function is initially called, `COBexClient` sets the `TRequestStatus` to `KRequestPending` to indicate that the request is in progress. If the `TRequestStatus` later completes with a value of `KErrNone`, an OBEX session has been successfully established with the OBEX server on the remote device. A completion with any other value indicates that an error has occurred, and no session has been established. The specific error value may indicate a particular type of failure (see Table 10.5). The `TRequestStatus` parameter is used in this same way for all overloads of `Connect()` and `ConnectL()`.

If authentication of the server is required during connection, there is a `ConnectL()` overload that takes a descriptor reference in addition to the `TRequestStatus&`. The descriptor contains the password that should be used to challenge the remote OBEX server during connection – that is, the password that we expect to see the remote OBEX server provide to us to prove its authenticity.

If there is data to be transferred during connection, there is a `ConnectL()` overload that takes a `COBexBaseObject&` in addition to the `TRequestStatus&`. The `COBexBaseObject` should have been

prepared with all the headers that need to be transferred during the connection (this is how a Target header is supplied for use during establishment of directed connections).

Note: the code that created the `COBexBaseObject` retains ownership, and so must ensure that it gets deleted when the Connect command is complete.

For situations where both authentication and connect data are required, there is an overload of `ConnectL()` that incorporates a descriptor containing a password, a `COBexBaseObject&` and a `TRequestStatus&`.

Following a successful connect using any of these overloads, the client is able to start issuing further commands over the established OBEX session.

```
void CMYObexClientApp::ConnectWithTarget()
{
    const TUint8 KTargetHeaderHi = 0x46;
    _LIT8(KTargetValue, "\xF9\xEC\x7B\xC4\x95\x3C\x11\xd2\x98\x4E\x52
                        \x54\x00\xDC\x9E\x09"); // Folder Browsing

    // iClient is an initialised member variable of type COBexClient*
    // iTargetHeaderObj is a member variable of type COBexBaseObject*

    delete iTargetHeaderObj;
    iTargetHeaderObj = NULL;

    iTargetHeaderObj = COBexNullObject::NewL();
    COBexHeader* targetHeader = COBexHeader::NewL();
    CleanupStack::PushL(targetHeader);
    targetHeader->SetByteSeqL(KTargetHeaderHi, KTargetValue());
    iTargetHeaderObj->AddHeaderL(*targetHeader);

    CleanupStack::Pop(targetHeader); // Ownership has been passed

    iClient->Connect(*iTargetHeaderObj, iStatus);
    SetActive();

    // Successful or unsuccessful completion of Connect command will
    // result in RunL of this object being called.
}
```

Example 10.9 Use of `Connect()`

Put

The `Put` function allows the OBEX client application to initiate the transfer of an object to the remote OBEX server. The function takes two parameters: a `COBexBaseObject&` and a `TRequestStatus&`. The `COBexBaseObject` must have been prepared with a source of body data (a file or buffer) and all the associated headers to be sent. The

TRequestStatus is used in the normal way, as with ConnectL(), to signal successful or unsuccessful completion of the Put() command.

```
void CMyObexClientApp::Put()
{
    // iClient is an initialised member variable of type CObexClient*
    // iFileObj is an initialised member variable of type
    // CObexBaseObject*

    iClient->Put(*iFileObj,iStatus);
    SetActive();

    // Successful or unsuccessful completion of Connect command will
    // result in RunL of this object being called.
}
```

Example 10.10 Use of Put

Get

Get() allows the OBEX client application to initiate the transfer of an object from the remote OBEX server. It takes parameters identical to those of Put(), but uses the CObexBaseObject& parameter in different way.

When Get() is called, the CObexBaseObject must have been prepared by the OBEX client application with all the headers that need to be transferred to the OBEX server during the specification phase of the Get command. CObexClient will ensure that the contents of the object are transferred during the specification phase. When the specification phase is complete, CObexClient resets the object (removes all its headers and body data) and then reuses it to store the object data returned from the OBEX server during the second phase of the Get command.

The TRequestStatus is used in the normal way to signal successful or unsuccessful completion of the Get command.

```
void CMyObexClientApp::GetByName()
{
    // iClient is an initialised member variable of type CObexClient*
    // iObject is an initialised member variable of type
    // CObexBaseObject*

    // Set up iObject to act as specification for the Get command
    // Specify name of object to be 'got' as 'contact.vcf'
    const TUint8 KNameHeaderHi = 0x01;
    _LIT(KFileName, "Contact.vcf");

    iObject->Reset();

    CObexHeader* nameHeader = CObexHeader::NewL();
    CleanupStack::PushL(nameHeader);
```

```

nameHeader->SetUnicodeL(KNameHeaderHi, KFileName());
iObject->AddHeaderL(*nameHeader);

CleanupStack::Pop(nameHeader); // Ownership has been passed

iClient->Get(*iObject, iStatus);
SetActive();

// Successful or unsuccessful completion of Get command will
// result in RunL of this object being called.
// Where the result is success, iObject will contain the retrieved
// object headers and body data.
}

```

Example 10.11 Use of Get

SetPath

The SetPath function allows the OBEX client application to initiate an OBEX SetPath command to the remote OBEX server. It takes two parameters: a TSetPathInfo& and a TRequestStatus&.

The TSetPathInfo class is defined with the scope of CObex class. It holds all the data that is transferred as part of a SetPath command, namely the flag byte, the constants byte and the value to be used in the name header. The OBEX client application must set the appropriate flags and (optionally) the name field in a TSetPathInfo object before calling the SetPath function (these fields can be set directly by user code, as they are defined as public member variables).

The TRequestStatus is used in the normal way to signal successful or unsuccessful completion of the SetPath command.

```

void CMyObexClientApp::SetPath()
{
    const TInt KDontCreateFolderFlag = 2;

    // iClient is an initialised member variable of type CObexClient*

    _LIT(KSubDirName, "symbianoscommsprogramming");

    CObex::TSetPathInfo info;
    info.iNamePresent = ETrue;
    info.iName = KSubDirName();
    info.iFlags |= KDontCreateFolderFlag;

    iClient->SetPath(info, iStatus);
    SetActive();

    // Successful or unsuccessful completion of SetPath command will
    // result in RunL of this object being called.
}

```

Example 10.12 Use of SetPath

Abort

The `Abort` function is used by the OBEX client application to abandon an ongoing OBEX command. The function has no parameters or return value, and has no effect if there is no OBEX command in progress.

If an OBEX command is in progress, the `COBexClient` will send an Abort request packet to the server in place of the next request packet in the ongoing OBEX command request–response exchange (`COBexClient` must obey the conversational protocol and wait for the OBEX server to respond before it can send another request). Once the server responds to the Abort request, the request that initiated the aborted OBEX command will have completion signalled with the value `KErrAbort`.

Except in error situations, the OBEX session will remain connected and synchronized following such an aborted command, which allows the OBEX client application to initiate further OBEX commands without disconnecting and reconnecting the session.

```
void CMyObexClientApp::Abort()
{
    // iClient is an initialised member variable of type COBexClient*
    // A command is in progress

    iClient->Abort();

    // The command will be aborted at the next opportunity, and the
    // TRequestStatus passed in when the command was started will
    // complete with KErrAbort.
}
```

Example 10.13 Use of `Abort`

Disconnect

Once all the necessary operations have been completed for a given OBEX session, the OBEX client application can tear down the OBEX session and the transport connection over which it runs. The `Disconnect()` function can be used to do this. It takes a `TRequestStatus&` parameter which is used to indicate completion of a successful or unsuccessful disconnection. If the disconnection is successful, the `TRequestStatus` will be completed with the value `KErrNone`.

If a disconnection is unsuccessful, the `TRequestStatus` will be completed with a value reflecting the response code from the server, and the transport connection and OBEX session will both remain connected. In fact, there is only one valid situation defined in OBEX specification where a disconnection is allowed to fail, and the Symbian OBEX implementation should preclude this from happening.

If such a problem should arise, it may be necessary to take more drastic action to shut down the transport connection and OBEX session. See 'Immediate shutdown of `COBexClient`'.

```

void CMyObexClientApp::Disconnect()
{
    // iClient is an initialised member variable of type COBexClient*

    iClient->Disconnect(iStatus);
    SetActive();

    // Successful or unsuccessful completion of Disconnect command will
    // result in RunL() of this object being called.
}

```

Example 10.14 Use of Disconnect

Immediate shutdown of COBexClient

In some circumstances, it may be necessary to immediately shut down the OBEX client and halt any ongoing activities. This may happen as the result of an error condition in the OBEX application, or as the result of the OBEX client application being required to shut down by the wider operating system, for example, when the phone is switched off.

This immediate shutdown can be achieved simply by deleting the COBexClient object. The ongoing transport connection, and with it the ongoing OBEX session, will be immediately disconnected.

Application writers should remember that they retain ownership of objects passed into COBexClient's Connect(), Put() and Get() functions, and remain responsible for deleting them.

GetPutFinalResponseHeaders

Some OBEX applications¹² require that the final response packet from the OBEX server should contain status information in addition to the response code, in the form of OBEX headers. When an OBEX client application is anticipating such headers at the end of a Put exchange, GetPutFinalResponseHeaders() can be used to access headers sent in the final Put response packet.

The function returns a const COBexHeaderSet& which can be inspected in the usual way using COBexHeaderSet::Find() or the MOBexHeaderCheck class. Ownership of the header set remains with the COBexClient.

OBEX command completion and error codes

When local errors (e.g., 'Out of memory') cause the completion of a request, the request will be completed with a standard Symbian system-wide error code.

When a particular command completes due to a successful (e.g., 'Success') or unsuccessful (e.g., 'Unauthorized') response from the remote

¹² Notably, the Bluetooth Basic Imaging Profile (BIP).

OBEX server, it can be useful for the client application to know precisely what the response code was.

The Symbian OS OBEX implementation uses a mapping to transform the response code that completes a given OBEX command to produce a Symbian-style error code (i.e., a `TInt` value of zero (`KErrNone`) or smaller). For historical reasons, some of the response codes are mapped to pre-existing Symbian system-wide error codes, while others are mapped individually to OBEX specific Symbian error codes, as illustrated in Table 10.5).

The other OBEX response codes are mapped one to one to a range of Symbian OBEX-specific error codes from `KErrIrObexRespSuccess` to `KErrIrObexRespDatabaseLocked`.

Note: the mapping of several OBEX response codes to a single Symbian error code may be restrictive in some cases where the OBEX client application requires the exact response code. In Symbian OS v9.2, `LastServerResponseCode()` will allow the OBEX client application to discover this response code following completion of an OBEX command. See section 10.2.10 for other additions in Symbian OS v9.2.

10.2.6 OBEX Server API

The OBEX server API in Symbian OS is defined by the classes `COBexServer` and `MOBexServerNotify`. An object of class `COBexServer` must be instantiated by the OBEX server application, and the `MOBexServerNotify` interface class must be inherited from and implemented by the OBEX server application. The class that implements `MOBexServerNotify` is very important as it responds to incoming events from remote OBEX clients, and so defines the behaviour of the OBEX service.

An OBEX server application instantiates its `COBexServer` using the `COBexServer::NewL()` factory function. At the time of creation, the application must have made choices on the transport over which OBEX should run, and (maybe) decisions about how the service should be advertised and how remote OBEX clients should access it.

Creation of `COBexServer` and selection of transport

As with `COBexClient`, there are two overloads of `COBexServer::NewL()` in Symbian OS v9.1 and three in v9.2. The v9.1 versions take as parameters the same `TOBexProtocolPolicy` and `TOBexProtocolInfo` derived structures as `COBexClient::NewL()`. For `COBexServer::NewL()`, however, the `TOBexProtocolInfo` derived structures are used slightly differently because in this case we are setting up a *passive* connection.

Table 10.5 OBEX response codes mapping to standard Symbian error codes

OBEX response code	Symbian error code
Continue Success Created Accepted	KErrNone
Bad Request Unauthorized Forbidden Not Acceptable Method Not Allowed	KErrArgument
Not Found	KErrNotFound
Timed Out	KErrTimedOut
Conflict	KErrInUse
Not Implemented	KErrNotSupported

Passive connections are a means to wait for some other entity (in this case a remote OBEX client) to initiate a connection. Once the other entity has initiated the connection, the passive connection spawns a ‘live’ connection over which the two entities can communicate.

At this point, it is worth considering the characteristics of the transport types and protocols to understand the differences in the way the transport is set up to provide a passive connection.

OBEX server connections over IrDA Transport set up is trivial for OBEX server applications running over IrDA, as `CObexServer` takes care of registering the IAS service advertisement and (by the use of the `KAutoBindLSAP` constant) selecting the TinyTP LSAP over which the OBEX server will run. This is illustrated in example code below.

```
__LIT8(KClassName, "OBEX");
__LIT8(KAttributeName, "IrDA:TinyTP:LsapSel");
TObexIrProtocolInfo aInfo;

info.iAddr.SetPort(KAutoBindLSAP);
info.iTransport = KObexIrTTPProtocol;
info.iClassName = KClassName;
aInfo.iAttributeName = KAttributeName;

iServer = CObexServer::NewL (aInfo);
```

Example 10.15 Setting up OBEX server with IrDA transport information

It should be noted that on return from `NewL()`, the transport has largely been prepared to accept incoming connections, but will not do so until `Start()` is called to activate the OBEX server (for IrDA, it is not until `Start()` is called that the IAS record is registered). See below for more information on `Start()`.

OBEX server connections over Bluetooth Transport set up for Bluetooth is somewhat more involved than for IrDA. Provided with an RFCOMM server channel number, `COBexServer` will create the passive Bluetooth connection, and wait for connection when subsequently told to `Start()`. However, this is not enough to allow remote devices to use the service. There are a number of other things that a Bluetooth server application must consider, specifically:

- Setting the major service class correctly in the class of device (CoD)
- Registering an SDP record
- Setting link security
- Device discoverability.

All of these topics are covered at greater length in Chapter 4. An OBEX-centric summary is included here for convenience.

Class of device – In Bluetooth, the class of device is a three-byte bit array that is returned from a device that is responding to an inquiry scan. Its purpose is to provide a quick, high-level view of what kind of device is responding to the inquiry (e.g., computer, phone) and what types of service the device supports. One of these service types is ‘Object Transfer’, and the bit corresponding to this service type should be set by the OBEX server application while the device is offering OBEX-based services over Bluetooth – however, see the discussion in Chapter 4 about possible problems with setting or unsetting this bit.

From v9.2, OBEX will set this bit appropriately when it starts up, so there is no need for the application to do anything about it (obviously the application still needs to register an SDP record though).

SDP – Each Bluetooth service passively waiting for a connection from some remote Bluetooth device must advertise itself in the SDP database so that remote Bluetooth devices can determine firstly that the service is supported and secondly how to connect to it. Specifically for OBEX services, this means over which RFCOMM server channel it runs.

`COBexServer` does not do SDP registration on behalf of an OBEX server application. This is for the same reason that the OBEX client does

not perform SDP searches on behalf of the OBEX client application; in addition to basic information such as server channel numbers, a Bluetooth service record in the SDP database may need to contain information that only an OBEX server application will be able to provide. Therefore, the OBEX server application is responsible for setting up its own SDP record, and tearing it down later when the service is closed down. See Chapter 4 for more details on how service records can be added and removed from the local service discovery database.

Link security – Security options which need to be applied to the Bluetooth link must be set in the `TObexBluetoothProtocolInfo`. Each Bluetooth link may optionally be authenticated, and when authenticated, optionally encrypted. It may also be marked as requiring explicit authorization from the user.

A description of the mechanisms involved is beyond the scope of this chapter, but the means to apply these link security options when OBEX runs over Bluetooth is straightforward.

```
// NB. iProtocolInfo is a TObexBluetoothProtocolInfo

TBTSserviceSecurity btServiceSecurity;
btServiceSecurity.SetAuthentication(EFalse);
btServiceSecurity.SetAuthorisation(ETrue);
btServiceSecurity.SetEncryption(EFalse);

iProtocolInfo.iAddr.SetSecurity(btServiceSecurity);
```

Example 10.16 Setting Bluetooth link security in OBEX

The values shown above would instate a low level of security for a service, specifically requiring the user to authorize a connection made to that service. This could be sufficient for a simple application used to beam a low-value object from one device to another in a one-off link operation (e.g., beaming a photograph from one phone to another).

Most services would require a higher level of security, requiring authentication to ensure the correct devices are connected together, and (normally when using an authenticated link) encryption to ensure that the data being transferred is not accessible to other devices able to ‘listen in’ on the Bluetooth radio link. Please note that requesting Bluetooth authentication for a link between two devices that have not previously authenticated each other will result in a process called ‘pairing’.

Device discoverability and ‘connectability’ – Discoverability and connectability are characteristics that describe the fundamental behaviour of a device when some other device is attempting to find it or connect to it using Bluetooth. These values must be set correctly, or the rest of the setting up above will be of no use!

When a device is offering an OBEX service, generally the device will need to be both discoverable and connectable. However, there may be some circumstances where the device only need be connectable, i.e., when the OBEX client device and the OBEX server device have a fixed relationship. For example, if the phone contains a database that needs to be regularly synchronized to the same PC; the phone may remain connectable but not discoverable, and the PC can store the phone's Bluetooth device address. The PC will then be able to connect to phone without first discovering it.

All this being said, control of discoverability and connectability is a device-wide consideration – OBEX server applications should not be trying to control discoverability or connectability of the local device. However, they may want to consider providing feedback to the user if there is some major problem that will affect their function, for example, Bluetooth is switched off!

Start and Stop

The `Start()` and `Stop()` functions of `COBEXServer` provide an 'on/off switch' for OBEX services.

Calling `Start()` achieves two things. Firstly, it allows the OBEX server application to provide `COBEXServer` with a reference to an instance of an `MOBEXServerNotify`-derived class. As mentioned above, the class implementing the `MOBEXServerNotify` interface responds to incoming events from remote OBEX clients, and so defines the behaviour of the OBEX service.

The class implementing the `MOBEXServerNotify` interface must be ready to start handling events because, secondly, `Start()` enables the transport to accept incoming connections. Once incoming connections commence, events will start to happen, triggering calls onto the `MOBEXServerNotify`-derived object.

`Stop()` can be called at any time once the `COBEXServer` has been started. `Stop()` will abandon any ongoing OBEX session activity by bringing down any transport link that has been established. No further connections to the OBEX service will be allowed until `Start()` is called again.

Note that `Stop()` is called as part of `COBEXServer`'s destructor, so it is valid to simply delete an instance of `COBEXServer` without calling `Stop()` first when rapid shutdown is required.

MOBEXServerNotify

`MOBEXServerNotify` defines a set of functions corresponding to events such as changes in transport status, and incoming OBEX request packets. The class that implements `MOBEXServerNotify` is required to respond

to these incoming OBEX events, usually taking service-specific action upon receiving them. For example, on being informed of an incoming Put request, the class must provide a prepared object in which to store the incoming data; what happens to that object once the transfer is complete is service-specific.

`MOBEXServerNotify` defines a synchronous interface; that is, the class implementing the interface must respond to an event by the time it returns program control to the `COBEXServer` class.

This is an occasionally inconvenient limitation that is addressed in the Symbian OS v9.2 release (see section 10.2.10).

Transport-related events and OBEX session

`COBEXServer` will keep the `MOBEXServerNotify`-derived object passed into `COBEXServer::Start()` informed of changes in the transport status and OBEX session status.

Connection `COBEXServer` will call `TransportUpIndication()` when the Bluetooth or IrDA transport link is established – this is before any OBEX request/response packets have passed over the link to establish the OBEX session. An OBEX server application can use this as an opportunity to initialize itself prior to receiving a Connect request.

Like `COBEXClient`, `COBEXServer` handles connection establishment automatically on behalf of the application, including any requested authentication and directed session activities. Once the OBEX session has been successfully established, `COBEXServer` will call `ObexConnectIndication()` to indicate this to the OBEX server application. `ObexConnectIndication()` passes a `TOBEXConnectInfo` as a parameter.

`TOBEXConnectInfo` contains a digest of useful information about the connected OBEX client application, including the Target header that the client application used during connection process. This can be used to discover the intent of the OBEX client application, and so modify behaviour of the OBEX server application accordingly. See section 10.2.8 on directed connections for more information on the subject of Target headers.

Once `ObexConnectIndication()` has been called, the OBEX server application should be prepared to respond to OBEX requests. Please note that the return value `TInt` of `ObexConnectIndication()` is a historical anomaly, and whatever value is returned here has no effect on the operation of `COBEXServer`. In this sense, `TransportUpIndication()` is purely an indication; it reports something that has already happened, and does not offer an opportunity to reject the connection.

Unsuccessful connection attempts (e.g., ones where authentication procedures have failed) are not reported to the OBEX server application. In these circumstances, the OBEX server application will ‘see’ a `TransportUpIndication()`, followed some short time later by a `TransportDownIndication()` as the client disconnects the transport link after its failure to establish an OBEX session.

Disconnection An OBEX client may disconnect a session with an OBEX server in one of two ways: either by sending an OBEX Disconnect command to bring down the OBEX session followed by a transport disconnection; or by simply disconnecting the transport. Both methods are valid according to the OBEX specification, although the first is considered more ‘polite’, and may in some circumstances be useful (it is not impossible that a Disconnect request may contain some header to indicate the reason for the disconnection for instance).

As a result, the OBEX server application may see an `ObexDisconnectIndication()` called on the `MOBexServerNotify`-derived object, followed by a `TransportDownIndication()`. Alternatively, the OBEX server application may see only a `TransportDownIndication()`. The OBEX server application may find it convenient to use `TransportDownIndication()` as an opportunity to release resources required only during an active OBEX session.

Handling Puts

PutRequestIndication Upon receiving the first Put request packet of a Put command, `COBexServer` will call `PutRequestIndication()` on the `MOBexServerNotify`-derived object passed into `COBexServer::Start()`. This allows the OBEX server application to provide a `COBexBufObject`¹³ into which the headers making up the object are to be stored. The address of the `COBexBufObject` is used as the return value for `PutRequestIndication()`. The OBEX server application retains ownership of the object.

If `NULL` is used as the return value of `PutRequestIndication()`, the Put response packet is sent to the client with a ‘Forbidden’ response.

Symbian OS v9.2 will make it possible to return a wider variety of response codes in this situation.

```
COBexBufObject* COBexFtpServer::PutRequestIndication()
```

¹³ Note specifically a `COBexBufObject`, rather than any `COBexBaseObject` derived class as might be expected. This is another historical anomaly that is addressed in Symbian OS v9.2.

```

{
    iTest.Printf(KObexPutIndication);

    // We will try and receive an object into the current directory

    TInt err=SetupReceiveObject();
    if(err)
    {
        // We couldn't set up the receiving object
        iTest.Printf(KObexPutRequestIndError, err);
        return NULL;
    }

    return iReceiveObject;
}

TInt CObexFtpServer::SetupReceiveObject()
{
    // Non-leaving equivalent to SetupReceiveObjectL()
    TRAPD(err, SetupReceiveObjectL());
    return err;
}

void CObexFtpServer::SetupReceiveObjectL()
{
    delete iReceiveObject;
    iReceiveObject=NULL;

    // iTempFileName is a TFileName
    // iFileSpec is a TParse containing the current path
    //
    iTempFileName.Copy(iFileSpec.DriveAndPath());
    iTempFileName.Append(KTempFileName);

    // iReceiveBuf is a preallocated CBufFlat
    //
    TOBexFilenameBackedBuffer bufDetails(*iReceiveBuf, iTempFileName,
                                         COBexBufObject::ESingleBuffering);

    iReceiveObject = COBexBufObject::NewL(NULL);

    iReceiveObject->SetDataBufL(bufDetails);
}

```

Example 10.17 Implementation of **MOBexServerNotify::PutRequestIndication()**

PutPacketIndication Once the MOBexServerNotify-derived object has returned the COBexBufObject pointer, the COBexServer calls PutPacketIndication() onto the same MOBexServerNotify-derived object.

This PutPacketIndication() call is made for the first and then each subsequent incoming Put request packet, and provides a means by which transfer progress may be tracked and perhaps indicated on a UI. It also offers an opportunity for the OBEX server application to respond to a Put request packet with an error response code, and

so brings the Put command to a premature halt. The return value of `PutPacketIndication()` is a Symbian style error code, and should always be `KErrNone` unless the server application wishes to stop the Put command. The same rules as described in ‘OBEX command completion and error codes’ apply to mapping between Symbian style error numbers and their corresponding OBEX response codes.

Because `PutPacketIndication()` is called for each request packet in an exchange, it is important that the processing that takes place in the implementation of the function is as lightweight and fast as possible; delays in processing a `PutPacketIndication()` call will delay the `COBexServer`’s response. Particularly when using small packet sizes, these delays can mount up to considerably impair the performance of the transfer.

PutCompleteIndication Once the final request packet of the Put command has been received, the `COBexServer` calls `PutCompleteIndication()` onto the same `MOBexServerNotify`-derived object. `PutCompleteIndication()` indicates that the transfer is over, and the object provided in response to `PutRequestIndication()` is ready to be inspected, processed or stored as appropriate.

`PutCompleteIndication()` allows the OBEX server application to respond to the final Put request packet with an error response code in the same way as `PutPacketIndication()` does. Although by this stage it is too late to stop the Put command, returning an error response code can give an indication to the OBEX client that the Put did not complete successfully, and why it was unsuccessful. In the example below for instance, the error returned when trying to rename the received file object is passed back to the OBEX client through the `err` variable.

```
TInt COBexFtpServer::PutCompleteIndication()
{
    // Check invariants
    // We should always have iReceiveObject set if we're
    // receiving PutCompleteIndication
    _LIT(KThreadName, "FTP Server");
    const TInt KNoReceiveObjectDefined = 1;
    __ASSERT_ALWAYS(iReceiveObject,
        User::Panic(KThreadName, KNoReceiveObjectDefined));
    // The Put is finished.
    // If the received object had a name header, we need to do a
    // number of things;
    // 1. Get the intended file name from the object
    // 2. Free up the file by deleting the receive object
    // 3. Rename the temporary received file to its correct name

    // Omission - Put with empty filename does not currently
    // result in file deletion

    TInt err = KErrNone;
```

```

COBexHeaderSet& hdrSet(iReceiveObject->HeaderSet());
hdrSet.First();
if(hdrSet.Find(KNameHeaderHi, *iHeader) == KErrNone)
{
    // The object's header set contains a Name header, and
    // iHeader now points to it.
    //
    // Making some assumptions here; that the target path
    // and contents of the name header actually do fit in a
    // standard TFileName structure.
    //
    TFileName targetFileName;
    targetFileName.Copy(iFileSpec.DriveAndPath());
    targetFileName.Append(iHeader->AsUnicode());

    // Now we have to delete the object to free up the file
    delete iReceiveObject;
    iReceiveObject = NULL;

    // Now attempt to rename the file to the target filename
    err = iFileServer.Rename(iTempFileName, targetFileName);
}
else
{
    // We've finished with the receive object. Free up the
    // memory.
    delete iReceiveObject;
    iReceiveObject = NULL;
}

return err;
}

```

Example 10.18 implementation of **MOBexServerNotify::PutCompleteIndication()**

SetPutFinalResponseHeaders Some OBEX applications will go further than simply using a response code, and include headers in the final Put response packet that provide more detailed information on the outcome of the Put command. This approach is supported by the Symbian OS OBEX APIs, through `GetPutFinalResponseHeaders()` on the OBEX client API, and through `SetPutFinalResponseHeaders()` on the OBEX server API.

The use of `COBexServer::SetPutFinalResponseHeaders()` is entirely optional, but is necessary when headers need to be sent in the final Put response packet. As a parameter, it takes a pointer to a `COBexHeaderSet` that contains all the headers that the OBEX server application wishes to return in the final Put response packet. It is important to note that `COBexServer` always takes ownership of this `COBexHeaderSet`, even when the function returns an error code. It can be called at any point during a Put command, although the most useful time to do so is generally during `PutCompleteIndication()`, when the outcome of the Put is known to the OBEX server application.

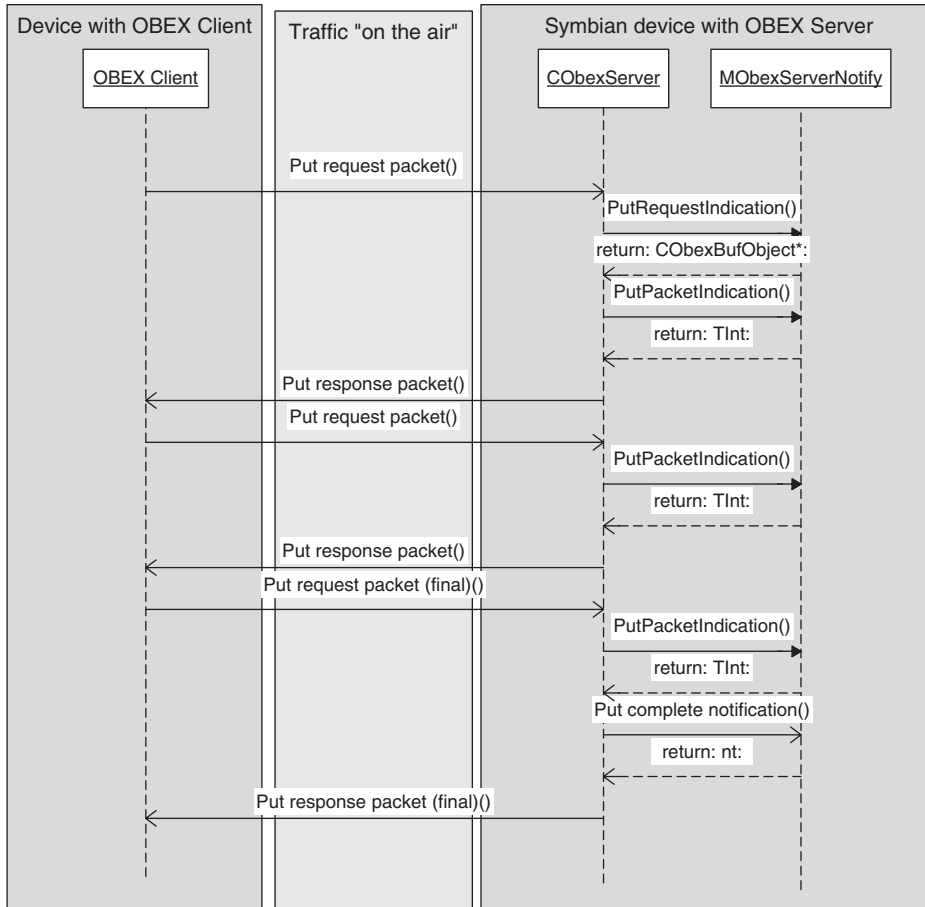


Figure 10.8 Indications to MObexServerNotify-derived class during Put command

Figure 10.8 shows the sequence of calls that occur during a typical Put operation.

Handling Gets

GetRequestIndication The sequence of events upon a Get command being initiated by an OBEX client is a little more complex than that of a Put command. This is because of the two-phase nature of the Get command. The object implementing MObexServerNotify is not informed of the ongoing exchange of Get request and Get response packets until the specification phase is over. The specification stage may itself contain several request–response exchanges, as can be seen in Figure 10.9.

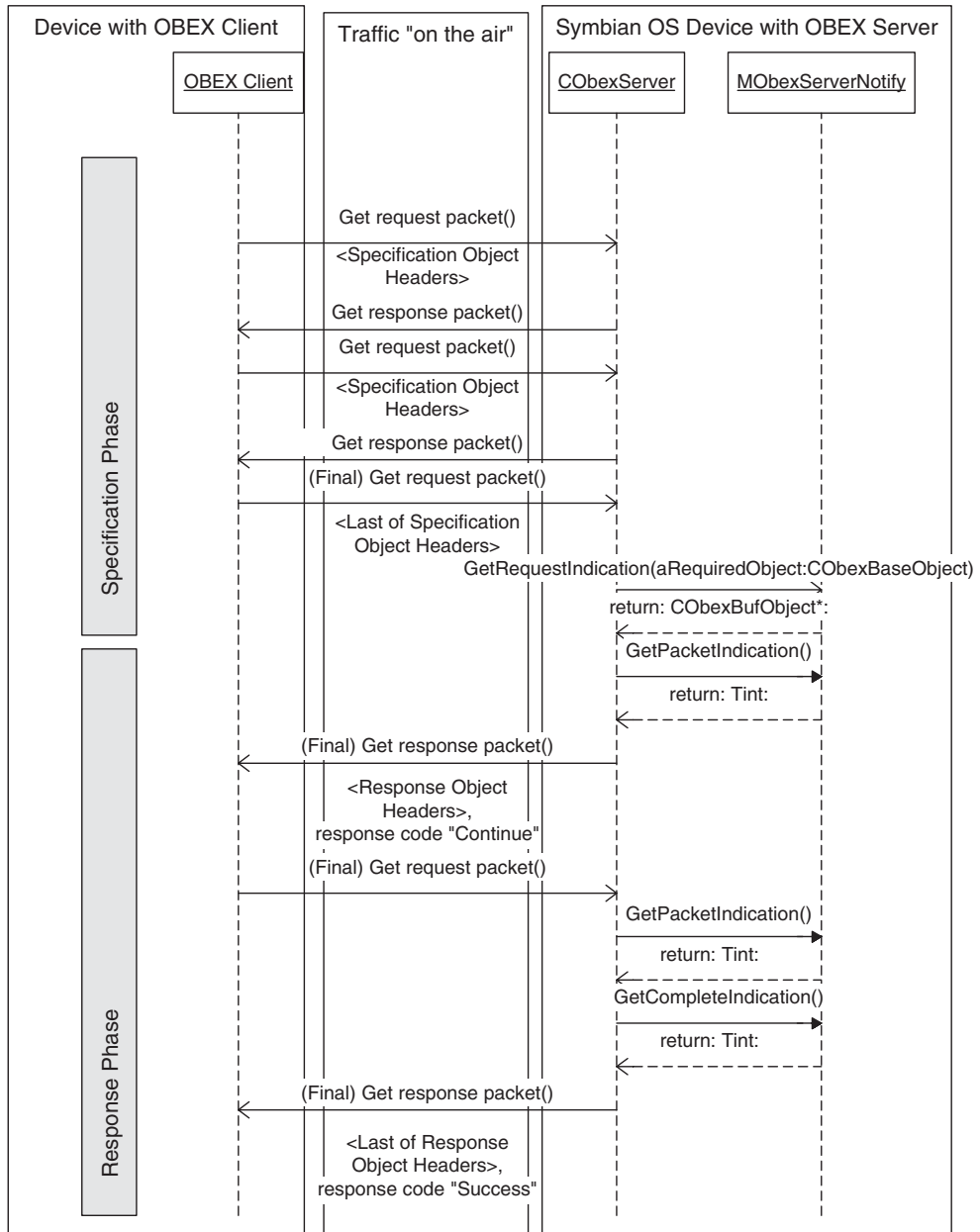


Figure 10.9 Indications to MobexServerNotify-derived class during Get command

Upon receiving the final Get request packet of the specification stage of a Get command, CObexServer will call `GetRequestIndication()` on the MObexServerNotify-derived object passed into

`COBexServer::Start()`. `GetRequestIndication()` passes a pointer to a `COBexBaseObject` as an argument. The `COBexBaseObject` in question represents the specification object transferred from the OBEX client, and so provides the OBEX server with the necessary information required to select the object to return to the OBEX client during the second phase (please note that the `COBexServer` retains ownership of the specification object). The OBEX server application initializes a `COBexBufObject` to represent the object to be returned, and then uses a pointer to this new object as a return value to `GetRequestIndication()`. Once this pointer has been returned, `COBexServer` will proceed with the second phase of the Get command and transfer the `COBexBufObject` to the OBEX client. The OBEX application retains ownership of the `COBexBufObject` passed as a return value throughout.

Please note that the specification object used by the `COBexServer` will not handle body data, so the specification must be contained in non-body headers.

If `NULL` is used as the return value of `GetRequestIndication()`, the Get response packet is sent to the client with a 'Forbidden' response.

Symbian OS v9.2 will make it possible to return a wider variety of response codes in this situation.

```
COBexBufObject* COBexFtpServer::GetRequestIndication(
    COBexBaseObject* aRequiredObject)
{
    iTest.Printf(KObexGetIndication);

    COBexBufObject* objectToSend = NULL;
    TInt err = KErrNone;

    COBexHeaderSet& hdrSet(iReceiveObject->HeaderSet());

    hdrSet.First();
    if(hdrSet.Find(KTypeHeaderHi, *iHeader) == KErrNone)
    {
        // The object's header set contains a Type header, and
        // iHeader now points to it.
        //
        if(iHeader->AsByteSeq() == KFolderListingType())
        {
            iTest.Printf(KObexFolderListingRequest);
            TRAP(err, PrepareFolderListingObjectL());
            if(!err)
            {
                objectToSend = iSendObject;
            }
        }
    }
}
```

```

if(!err && (objectToSend == NULL) )
{
    hdrSet.First();
    if(hdrSet.Find(KNameHeaderHi, *iHeader) == KErrNone)
    {
        // The object's header set contains a Name header,
        // and iHeader now points to it.
        // This indicates a request to fetch a file.
        iTest.Printf(KObexFileRequest, &(TPtrC(iHeader->AsUnicode())));

        HBufC* bufPtr = HBufC::New(KMaxPath+KMaxFileName);

        if(!bufPtr)
        {
            // Couldn't allocate buffer for filename
            return NULL;
        }

        TPtr fileToSend = bufPtr->Des();
        fileToSend.Append(iFileSpec.DriveAndPath());
        fileToSend.Append(iHeader->AsUnicode());

        TRAP(err, iSendObject->SetDataBufL(fileToSend));
        if(!err)
        {
            objectToSend = iSendObject;
        }
        delete bufPtr;
    }
}

return objectToSend;
}

```

Example 10.19 Implementation of **MOBexServerNotify::GetRequestIndication()**

GetPacketIndication Once the MOBexServerNotify-derived object has returned the COBexBufObject pointer, the COBexServer calls GetPacketIndication() onto the same MOBexServerNotify-derived object.

The GetPacketIndication() call is made for the first and every subsequent incoming Get request packet. As with PutPacketIndication(), it provides a means to track progress, and a means to respond to a Get request packet, and so brings the Get command to a premature halt.

All the same considerations as with PutPacketIndication() apply to GetPacketIndication(), including the advisability of keeping the implementation lightweight and fast so as not to impair transfer performance.

GetCompleteIndication Just before the final response packet is sent to the OBEX client, COBexServer calls GetCompleteIndication() on the same MOBexServerNotify-derived object. This

provides the OBEX server application with a final chance to return an error response code for the Get command. Since the burden of inspecting, processing or storing the object is on the OBEX client in this case, usually the implementation of this function simply returns `KErrNone`.

Handling *SetPath*

A `SetPath` command always consists of a single request packet and corresponding response packet. Upon receiving a `SetPath` request packet, `COBexServer` will call `SetPathIndication()` on the `MOBexServer` `Notify`-derived object passed into `COBexServer::Start()`.

`SetPathIndication()` passes two arguments: a `COBex::TSetPathInfo` reference and a descriptor reference. All the necessary information is encoded the `TSetPathInfo` object, and the descriptor is unused at this time.

`TSetPathInfo` consists of fields corresponding to the additional header information encoded into a `SetPath` request packet (the `Flags` and `Constants` fields defined in section 3.3.6 of the OBEX specification), plus the contents of the `Name` header if a `Name` header was included in the request packet. The `SetPath` flags are very simple; bit 0 is set if the OBEX server is expected to ‘back up’ by a level before acting on the rest of the `SetPath` request information. Bit 1 is set to indicate that the OBEX server should not create a directory of the name provided in the `Name` header if such a directory doesn’t already exist.

A `TBool` member variable `iNamePresent` can be used to determine whether a `Name` header was included in the `SetPath` request before accessing the `iName` descriptor member variable (this differentiates between the case where no name was present and the case where a `NULL` name was present). An illustration of how to access these fields is included in the code example below.

The OBEX server application can take whatever action is appropriate upon receiving a `SetPathIndication()` before returning an error code to indicate the outcome of the `SetPath` command to the OBEX client application. Mapping between the Symbian error codes and the OBEX response codes is achieved in the usual way (see Table 10.5). Appropriate action may mean navigating to a new directory in an actual file system, or something more abstract such as navigating to a conceptual ‘location’ offering a different subset of service behaviour (for example, see the `IrMC` specification or the `Bluetooth Phone Book Access Profile`).

```
TInt COBexFtpServer::SetPathIndication(const COBex::TSetPathInfo&
                                     aPathInfo, const TDesC8& /*aInfo*/)
{
    TBool backUpDir = aPathInfo.Parent();
    TBool dontCreateDir = aPathInfo.iFlags&KOBexDontCreateDir;
    TBool namePresent = aPathInfo.iNamePresent;
```

```

// Log the setpath indication
iTest.Printf(KObexSetPathIndication);
iTest.Printf(KObexSetPathBefore, &(TPtrC(iFileSpec.DriveAndPath())));

if(namePresent)
{
    iTest.Printf(KObexSetPathIndInfo1, &aPathInfo.iName);
}

iTest.Printf( KObexSetPathIndInfo2,
              (backUpDir?&KTrueText():&KFalseText()),
              (dontCreateDir?&KTrueText():&KFalseText()));

// We want this to be an atomic operation, so work on a
// copy of the current directory spec in case something
// goes wrong.
TParse curDir = iFileSpec;

TInt err = KErrNone;

// Back up if requested...
if(backUpDir)
{
    err = curDir.PopDir();
}

// Move down a directory if requested...
if(!err && namePresent)
{
    if(aPathInfo.iName == KNullDesC)
    {
        // Special case where path is to be reset to root
        curDir.Set(KStartDirectory, NULL, NULL);
    }
    else
    {
        // Check whether the named directory exists as
        // sub-directory of current directory
        TBool dirExists = EFalse;
        err = DirExists(curDir.DriveAndPath(), aPathInfo.iName,
                       dirExists);

        if(!err && !dirExists)
        {
            // Directory doesn't exist
            //
            if(!dontCreateDir)
            {
                // Directory doesn't exist, but we're
                // allowed to create it
                //
                err = iFileServer.MkDir(curDir.DriveAndPath());
                dirExists = err?EFalse:ETrue;
            }
        }
        else
        {
            // Directory doesn't exist, we're not
            // allowed to create it

```



```

        //
        err = KErrIrObexRespForbidden;
    }
}

// Add directory if it exists or we've been
// allowed to create it
if(!err && dirExists)
{
    err = curDir.AddDir(aPathInfo.iName);
}
}

if(!err)
{
    // Finally, we get to change the current directory, as
    // stored in iFileSpec so long as we haven't hit any
    // errors.
    iFileSpec = curDir;
}

// Log the outcome of the setpath
iTest.Printf(err ? KSetPathError() : KSetPathSuccess(), err);
iTest.Printf(KObexSetPathAfter, &(TPtrC(iFileSpec.DriveAndPath())));

return err;
}

```

Example 10.20 Implementation of **MOBexServerNotify::SetPathIndication()**

Handling Abort

When the OBEX client application sends an Abort request packet to abandon some other ongoing OBEX command, COBexServer will call AbortIndication() on the MOBexServerNotify-derived object passed into COBexServer::Start().

In response to this indication, the OBEX server application needs to only take any action necessary to prepare itself to accept another OBEX command. Since the protocol definition of OBEX permits no other response but 'Success', AbortIndication()'s return value is void, and the COBexServer ensures that the response packet is always sent with a 'Success' response code.

Handling errors

When a problem occurs with an OBEX session that requires disconnection, COBexServer will call ErrorIndication() on the MOBexServerNotify-derived object passed into COBexServer::Start(). ErrorIndication() passes a TInt parameter containing a Symbian error code to indicate the nature of the problem. The function is purely informational, and has a void return value.

Once the OBEX server application has been notified of the error, `COBexServer` will bring the transport down. As a result, a `TransportDownIndication()` may be expected to follow soon after the `ErrorIndication()`.

10.2.7 OBEX Sessions with Authentication and Directed Connections

Authentication in OBEX has been touched on above in relation to the Connect command and authentication challenge and authentication response headers, and directed connections in relation to the Target, Who and Connection ID headers. The Symbian OS OBEX implementation has support for authentication, and to a limited degree, directed connections also. In this section we will learn how to make use of these concepts in Symbian applications.

Authentication

Symbian OS supports OBEX authentication during the processing of OBEX Connect commands only; authentication for individual commands (such as Put or Get) is not supported. This applies to both OBEX server and OBEX client implementations.

If an OBEX client application or an OBEX server application wishes to authenticate a remote peer application while setting up an OBEX session, it must provide a password to use for the challenge. The same password must be provided to the remote peer application in order for it to successfully respond to the challenge, and so become authenticated. Note that the password is never transmitted over the link, but rather a hash value is sent based on (amongst other things) the password.

If the local OBEX application might be challenged by some remote OBEX application, the OBEX application must be prepared to provide a response password on request – this will usually require some user interaction.

During the establishment of an OBEX session, it is possible to have the OBEX client authenticate the OBEX server, or the OBEX server authenticate the OBEX client, or both, or neither. The sequence of events in protocol terms for each of these possibilities is described in ‘Connect’ in section 10.1.4.

The means by which Symbian-based OBEX client and server applications may provide the challenge and response passwords to the OBEX stack is covered in the next sections.

OBEX server authentication by an OBEX client

We have already seen how a password may be passed as a parameter to a `COBexClient` when calling `ConnectL()`. This password is then

used to challenge the OBEX server to which the `COBexClient` is trying to connect.

OBEX client authentication by an OBEX server

Because they cannot predict when an incoming connection attempt may occur, OBEX server applications implemented in Symbian OS should provide the challenge password before calling `Start()` on `COBexServer`. They do this by calling `COBexServer::SetChallengeL()`, passing in a descriptor containing the password. This password is used to challenge any OBEX client that attempts to connect to the OBEX server.

Responding to authentication challenges

Both OBEX server applications and OBEX client applications in Symbian OS use exactly the same API mechanism to be informed of and respond to incoming authentication challenges. This is possible because `COBexClient` and `COBexServer` are both derived from the same base class, `COBex`.

To enable OBEX applications to be informed that they are being challenged for authentication, they must implement a class called `MOBexAuthChallengeHandler`. The OBEX application must call `COBex::SetCallBack()`, passing in a reference to an instance of a class derived from `MOBexAuthChallengeHandler`.

`MOBexAuthChallengeHandler` is a simple mixin class, defining a single pure virtual function `GetUserPasswordL()`. As an argument, `GetUserPasswordL()` passes a descriptor containing the authentication realm (the realm is an optional field intended for display to users, often blank in practice).

When any authentication challenge is directed to the OBEX application, the `COBexClient` or `COBexServer` will call `GetUserPasswordL()` on the provided `MOBexAuthChallengeHandler`-derived object to indicate that a password is required to complete authentication.

On being challenged, an OBEX application will normally have to perform some user interaction to elicit the password from the user. The OBEX application is not required to provide the response password by return of the `GetUserPasswordL()` function; rather, once the user interaction is complete, the application calls into the `COBexClient` or `COBexServer` using the `COBex::UserPasswordL()` function. However, if the response password is already known to the application, `COBex::UserPasswordL()` can be called back immediately from within `GetUserPasswordL()`. `UserPasswordL()` takes a descriptor parameter which contains the response password. Once `UserPasswordL()` has been called following a `GetUserPasswordL()` upcall, the challenge will be responded to, and the connection process will proceed according to the normal protocol rules.

If an OBEX application is challenged for authentication and it has not provided a `MOBEXAuthChallengeHandler`, the authentication (and consequently the connection) will fail.

10.2.8 Directed Connections

In the OBEX specification, directed connections provide a way in which requests and responses for an OBEX session can be routed to a particular OBEX server that can handle the requirements of the OBEX client in a transport-independent way. This also provides a means by which multiple directed OBEX sessions can be multiplexed over a single underlying transport connection (see Figure 10.10).

Symbian OS implements a weaker form of directed connections, where the OBEX server can determine the requirements of the OBEX client (based on a Target header sent with a Connect request packet). The OBEX server may then reject the connection if it cannot, or will not, service the OBEX client's needs. No OBEX level routing or multiplexing is supported (although arguably a single OBEX client can connect to a server that offers multiple services – it simply can't be connected to more than

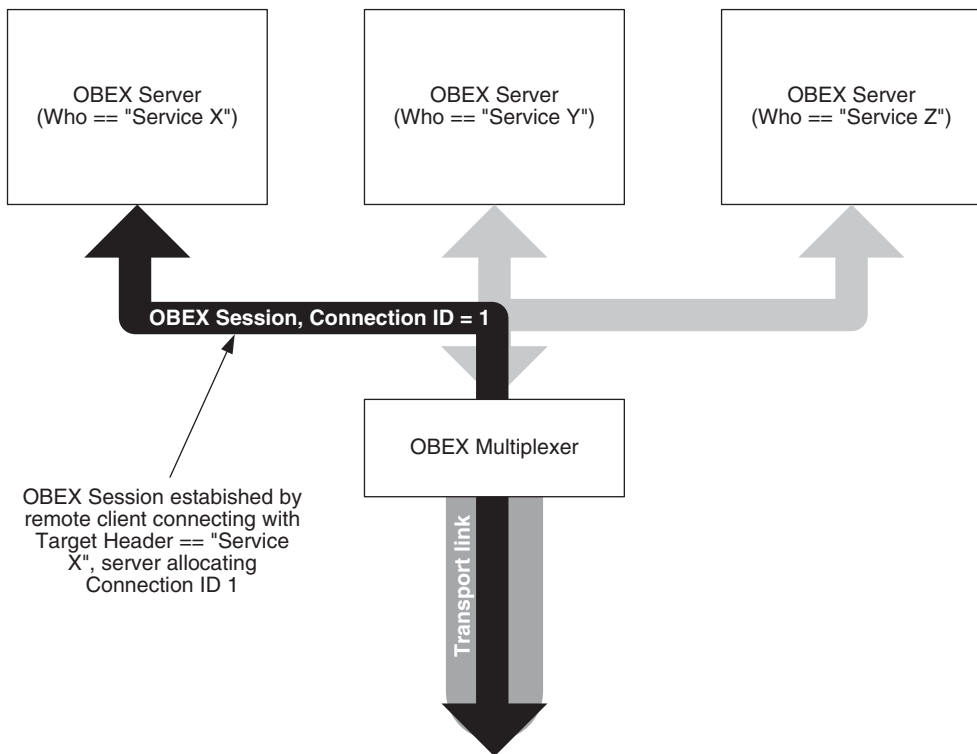


Figure 10.10 OBEX directed connections

one of these services simultaneously). The lack of full-directed connection support does not generally cause problems because the transports over which OBEX runs (Bluetooth and IrDA) provide their own methods of service discovery, routing and multiplexing (see Figure 10.11).

Directed connections are tied up with the use of the Target header, the Who header and the Connection ID header, as described in 'Target, Who and Connection ID headers' in section 10.1.2.

OBEX client and directed connections

The role of OBEX clients in Symbian OS in forming directed connections is quite small. They can specify a target header to be sent during connection by creating an object containing a Target header and passing this object into `Connect()` or `ConnectL()` used to initiate the connection. Once the transport link to the OBEX server has been established, the Target header is transmitted to the OBEX server as part of the Connect request packet.

If the connection attempt is successful, the OBEX client will receive a Connect response packet containing a Who header and a Connection ID header. The Who header indicates the identity of the service provided by the OBEX server (which will normally match the Target header), and the Connection ID uniquely identifies the session established between

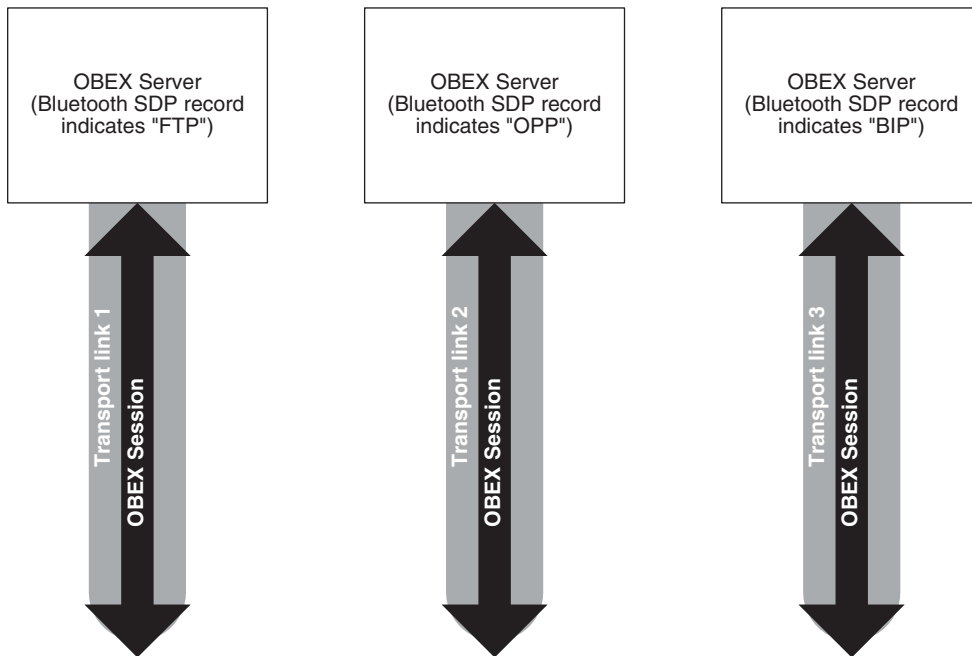


Figure 10.11 OBEX using transport multiplexing

the OBEX client and OBEX server. Symbian OS applications do not need to concern themselves with these details, however, as the Symbian OS OBEX implementation takes care of these headers automatically. If an OBEX client application does wish to know the Who identity of the remote server, it can use `COBex::RemoteInfo()` to retrieve details of the remote server (including the value of the Who header) collected during the connection process. The Connection ID is unlikely to be of use to OBEX applications, but may be extracted by the OBEX server from the header set of any object transferred from OBEX client to an OBEX server over a directed connection.

OBEX server and directed connections

In Symbian OS the options open to an OBEX server application relating to directed connections are a little wider. The OBEX server application may choose one of three methods to apply to processing the Target header during connections, each one being useful in different circumstances.

The three methods are identified in an enumeration `COBexServer::TTargetChecking` as shown in Table 10.6.

(**Note:** ‘Correct’ in this context means that the Target header matches the OBEX server’s ‘Who’ value.)

The OBEX server application can choose the method required by calling `COBexServer::SetTargetChecking()` before calling `COBexServer::Start()`.

The behaviour of each method is described below, along with an example of a situation in which that method might be considered most appropriate.

`ENoChecking` is the most promiscuous of the behaviours: when using this method, `COBexServer` will accept all connection requests regardless of whether a Target header is included or not included in the Connect request packet(s), and regardless of the value of the Target header. An OBEX server application choosing to exhibit this behaviour

Table 10.6 Symbian OS OBEX target header checking methods

Method	OBEX server behaviour
<code>ENoChecking</code>	Always allow the connection, regardless of presence, absence or value of Target header
<code>EIfPresent</code>	Allow the connection if the Target header is absent, or present and the correct value (this is the default behaviour for the Symbian OS OBEX server component)
<code>EAlways</code>	Allow the connection if the Target header is present and the correct value

might be some kind of multi-purpose application that advertises multiple services over the same transport link, but performs different services depending upon the Target header provided by the OBEX client.

`ElIfPresent` is the default behaviour for the Symbian OS OBEX server. When using this method, the OBEX server component checks the Target header only if it has been sent as part of a Connect request. If the Target header is present and incorrect, the connection is rejected. This method relies on the service discovery, routing and multiplexing mechanisms of the underlying transport working correctly, and the OBEX directed connection mechanism is simply used as a further level of checking. Unless multiple OBEX services are being advertised over a single transport link, this behaviour is normally entirely adequate.

`EAlways` is the most strict of the behaviours. In this case, connections are only accepted where the Target header is present and correct in the Connect request. This behaviour is best suited to specialized OBEX applications whose specification insists on the use of Target headers.

10.2.9 Platform Security Requirements of OBEX

As mentioned in section 10.2.3 above, the Symbian OS OBEX implementation runs in the same thread as the OBEX application code which hosts it. Symbian OS platform security is applied on process boundaries (and during DLL loading) only, so OBEX itself does not place any platform security requirements on applications that use it.

However, OBEX makes use of transport technologies that do place platform security requirements on their clients – in this respect, OBEX and OBEX applications count as clients of these transport technologies. For Bluetooth and IrDA, this means the use of `LocalServices` capability. More information on the platform security requirements of these technologies can be found in their respective chapters.

10.2.10 Changes in Symbian OS v9.2

Symbian OS v9.2 includes a number of small but significant changes to the OBEX APIs. These changes do not break source or binary compatibility with v9.1, so code written and binaries built against v9.1 will continue to build and work. The most important of these changes are described below.

Asynchronous server API

As mentioned above, the synchronous nature of the interface defined by `MOBexServerNotify` is occasionally inconvenient when asynchronous processing is required to decide how to respond to a request indication.

For this reason, in Symbian OS v9.2, a new interface has been defined to remove this restriction. The new interface consists of a class `MOBexServerNotifyAsync` and a group of new functions for `COBexServer`.

`MOBexServerNotifyAsync` looks very similar indeed to `MOBexServerNotify`, but on close inspection you may notice that some of the functions differ with regards to return parameters, coloured grey in Table 10.7.

The functions for which the return value disappears may now be used asynchronously. What this means in practical terms is that the application code that implements the functions can return program control to the `COBexServer` (note the `void` return parameters) and do whatever processing they need to do asynchronously before finally giving the

Table 10.7 Comparison of `MOBexServerNotify` and `MOBexServerNotifyAsync`

Function	Return value in <code>MOBexServerNotify</code>	Return value in <code>MOBexServerNotifyAsync</code>
ErrorIndication	void	void
TransportUpIndication	void	void
TransportDownIndication	void	void
ObexConnectIndication	TInt	void
ObexDisconnectIndication	void	void
PutRequestIndication	COBexBufObject*	void
PutPacketIndication	TInt	Tint
PutCompleteIndication	TInt	void
GetRequestIndication	COBexBufObject*	void
GetPacketIndication	TInt	Tint
GetCompleteIndication	TInt	void
SetPathIndication	TInt	void
AbortIndication	void	void
CancelIndicationCallback	Not present in <code>MOBexServerNotify</code>	void

`COBexServer` the object or response value it needs to proceed with the OBEX exchange.

The way in which the OBEX server application does this is by using the appropriate function out of several new functions added to `COBexServer` in Symbian OS v9.2.

To complete processing of a `PutRequestIndication()` or a `GetRequestIndication()` upcall, the OBEX server application must call one of the following three new functions on `CobexServer`:

```
TInt RequestIndicationCallback(COBexBaseObject* aObject);
TInt RequestIndicationCallbackWithError(TObexResponse aResponseCode);
TInt RequestIndicationCallbackWithError(TInt aErrorCode);
```

The first function is used when the OBEX server application wishes to proceed with the transfer. One of the second and third functions is used when the OBEX server application wishes to abandon the transfer with an error code. The second function allows the OBEX server application to directly specify an OBEX error code to be used in the response packet, while the third function takes a Symbian error code and maps it to an OBEX response code in the usual way.

To complete processing of a `PutCompleteIndication()`, a `GetCompleteIndication()` or a `SetPathIndication()`, the OBEX server application must call one of the following two new functions on `COBexServer`:

```
TInt RequestCompleteIndicationCallback(TObexResponse aResponseCode);
TInt RequestCompleteIndicationCallback(TInt aErrorCode);
```

These two overloads of `RequestCompleteIndicationCallback()` correspond to the two overloads of `RequestIndicationCallbackWithError()` in the treatment of their arguments.

Once these 'callback' functions have been called appropriately, processing of the request proceeds just as in Symbian OS v9.1. Care should be taken to call the functions only when appropriate – that is after a request indication upcall or a request complete indication upcall – because `COBexServer` will panic if these functions are abused.

The other significant additional function in `MOBexServerNotifyAsync` is `CancelIndicationCallback()`. This function is used by `COBexServer` to inform an OBEX server application that a problem has occurred and the callback is no longer necessary. When `CancelIndicationCallback()` is called, the OBEX server application must heed this information and not call back, or a panic will result.

Readers should note that the change of return parameter on `ObexConnectIndication()` from `TInt` to `void` is not due to an attempt to somehow make that indication asynchronous, but rather simply to

remove an unnecessary parameter from the function (the returned `TInt` specified in the `MOBEXServerNotify` class is a historical anomaly, and has no effect on processing of the Connect requests).

Setting timeouts on Client commands

`COBEXClient` has a new member function in Symbian OS v9.2 to enable OBEX client applications to protect against OBEX servers that become unresponsive during OBEX sessions.

An OBEX client application that calls `COBEXClient::SetCommandTimeout()` prior to initiating an OBEX command will be protected against unresponsive OBEX servers by a timer that starts when the OBEX request packet has been sent to the OBEX server. If the timer expires before a response packet has been forthcoming from the OBEX server, the `TRequestStatus` associated with the OBEX command will complete a specific timeout error, `KErrIrObexRespTimedOut`, and the transport link will be brought down.

This method of detecting unresponsive OBEX servers is much better than setting an overall timeout for the entire operation as it is often difficult to predict how long a given command will take to complete, while a relatively short timeout can be applied to individual request-response exchanges. The value of the timeout should be set large enough that the OBEX server will have an opportunity to build the first response packet, but not so large that the user is left waiting too long for feedback – around five seconds would be a good starting value.

Retrieving the most recent response code from an OBEX server

Due to a historical anomaly in the mapping of OBEX response codes to Symbian error codes used by the `COBEXClient` API, it is not always possible to determine the exact response code that was returned in a response packet by an OBEX server following a request packet. This limitation is removed in Symbian OS v9.2 with the addition of `COBEXServer::LastServerResponseCode()`. The function returns the full, untranslated response code from the most recent response packet sent from the server. Use of this function must be restricted to when the OBEX client application is sure that a response packet has been returned by the OBEX server, or a panic will result.

10.3 Summary

In this chapter we've seen:

- OBEX is intended as a simple, compact protocol intended to serve a similar purpose to HTTP on mobile or resource constrained devices

- it is a request-response based protocol operating between a client and a server. It uses Headers encoded within OBEX packets to convey data between the two
- OBEX offers a convenient abstraction for transferring objects (often files) between devices
- it can work over multiple different transports including Bluetooth and IrDA, offering potential for reuse of the same core OBEX application code over more than one transport.

11

HTTP

11.1 HTTP Overview

HTTP is a communication mechanism for transferring information between devices, defined by the IETF. Originally designed for transferring web pages, it has expanded beyond its original role into a general transport protocol between devices. This has been greatly helped by the fact that HTTP is allowed through many firewalls where other IP-based protocols are blocked.

The version of HTTP now in use by most web servers is HTTP 1.1. HTTP uses a client-server model and runs at the level above TCP. The most important standard, RFC2616, describes the makeup of messages sent between an HTTP client and an HTTP server. The client sends *requests* to the server and receives *responses*.

11.1.1 HTTP APIs

Symbian OS provides an HTTP client stack known as the *HTTP framework*. This was first introduced in Symbian OS v7.0. This chapter describes how to use the HTTP framework, starting with a simple Get command and then looking at more powerful customization of the framework. All protocol-defined request methods (including Get and Post) are supported.

Applications using the framework don't need to know about HTTP protocol details, allowing them to concentrate on creating and processing content. The framework provides a set of APIs – this chapter illustrates how you can use them.

Users of the framework in Symbian OS include the Java HTTP implementation, SyncML, and the Online Certificate Status Protocol (OSCP). Example applications that might use the HTTP framework include web browsers or a download manager.

An example application that allows uploading of images to the web-based photo sharing service Flickr, will be used to illustrate API usage. This provides a API – `CFlickrRestAPI` – which is used by the MTM example in Chapter 9 to extend the SendAs service to allow uploading of images to Flickr,

In addition to standard HTTP 1.1 services, the following functionality is supported by the framework:

- compatibility with HTTP 1.0 servers
- authentication, both basic and digest (defined in RFC 2617)
- secure connections (RFC 2818) – triggered by specifying `https` as the URI scheme in the request URI. The secure connection will be made automatically.

What is *not* supported:

- no caching support is provided by default, although some UI platforms provide filters that implement caching
- cookie management
- content encoding (or decoding). Again, some UI platforms may deliver filters that implement content encoding and decoding.

The architecture of the HTTP framework

Figure 11.1 shows the high-level architecture of the HTTP framework, including the filters that can be used to extend or modify the framework's behaviour. We describe the client APIs first; then move on to the use of the supplied filters, as well as the various parameters that can be used to configure a connection; then finally finish off by talking about how to create your own filters to load into the framework.

We also discuss the use of the Stringpool by the HTTP framework. The Stringpool is a mechanism for efficiently storing and comparing strings by keeping one central copy of the string per thread, and allowing clients to pass a compact representation of the string around (internally this is represented as an integer, but the classes for accessing strings hide this detail from the API user). Use of the Stringpool is particular beneficial for protocols that use a lot of well-known and static text strings, such as HTTP. Further details are given in section 11.8.

11.2 Getting Started: Creating a Session

Although HTTP is a stateless protocol, the framework supports the notion of a session, during which multiple HTTP transactions can take place.

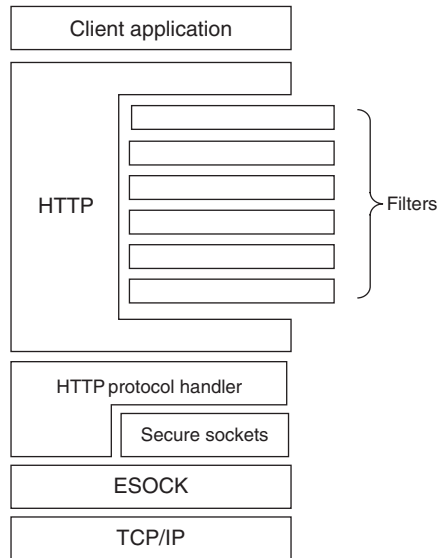


Figure 11.1 High-level architecture of the HTTP framework

For example a web browser would use one session but is likely to have multiple HTTP transactions, some of which may take place concurrently.

First, we need to open a session using the session API—`RHttpSession`. `RHttpSession` has an overhead in both memory use and creation time, so typically you should only create one session in an application. Applications that wish to perform HTTP transactions over multiple network connections should note, however, that each HTTP session is bound to a single `RConnection`, so they need one `RHttpSession` per network connection they wish to use.

Here are the most interesting methods on `RHttpSession`:

```
class RHttpSession
{
public:
    IMPORT_C void OpenL();
    IMPORT_C void OpenL(const TDesC8& aProtocol,
        MHTTPFilterCreationCallback* aSessionCallback);
    IMPORT_C void Close();
    IMPORT_C RHTTPTransaction OpenTransactionL(const TUriC8& aURI,
        MHTTPTransactionCallback& aCallback, RStringF aMethod = RStringF());
    IMPORT_C RStringPool StringPool() const;
    IMPORT_C RHTTPConnectionInfo ConnectionInfo() const;
    inline RHTTPFilterCollection FilterCollection() const;
    IMPORT_C RHTTPHeaders RequestSessionHeadersL();
    IMPORT_C RHTTPHeaders ResponseSessionHeadersL();
};
```

We create a session by calling `OpenL()`. Within the session a client makes requests *to*, and monitors responses *from*, different HTTP servers. Example 11.1 shows the session being configured for use with the example Flickr upload service.

```
void CFlickrRestAPI::ConstructL()
{
    // Open RHTTPSession with default protocol ("HTTP/TCP")
    iSession.OpenL();

    // initialize handles
    User::LeaveIfError(iSocketServ.Connect());
    User::LeaveIfError(iConnection.Open(iSocketServ));

    // set them for use with open http session represented by iSession
    RStringPool strP = iSession.StringPool();
    RHTTPConnectionInfo connInfo = iSession.ConnectionInfo();
    connInfo.SetPropertyL(strP.StringF(HTTP::EHttpSocketServ,
        RHTTPSession::GetTable()), THTTPHdrVal(iSocketServ.Handle()));
    TInt connPtr = reinterpret_cast<TInt>(&(iConnection));
    connInfo.SetPropertyL(strP.StringF(HTTP::EHttpSocketConnection,
        RHTTPSession::GetTable()), THTTPHdrVal(connPtr));

    // Setup the usual headers and indicate that the body will be multipart

    RHTTPHeaders httpHeader = iSession().RequestSessionHeadersL();
    SetHeaderL(httpHeader, HTTP::EUserAgent, KUserAgent);
    SetHeaderL(httpHeader, HTTP::EAccept, KAccept);
    SetHeaderL(httpHeader, HTTP::EContentType, KMultipartContentType());
    ...
}
```

Example 11.1 Creating a session

In Example 11.1 there are a few more details which need explaining:

1. `RequestSessionHeadersL()` is used to get the session request headers. These headers, once set, will be applied to *all* requests sent in the session.
2. We are setting some session properties so the session will use our network connection (`iConnection`).

11.2.1 Session Properties

The framework has properties that can be used to modify the session configuration. These can be set by the client and will be applied to all activities within a session.

A particular property is set by calling `CHttpConnectionInfo::SetProperty()`. The existing value of a property can also be retrieved using `RHTTTPPropertySet`.

The different property values, which alter the behavior of the framework in various ways, are defined in `RHTTPConnectionInfo.h`.

By default the framework will create a network connection for you. This will be owned by the framework and be created using the settings for the default access point in `CommsDat`. This may, depending on the configuration, trigger a user prompt asking them to select a network connection. You can override this behavior by providing your own `RConnection` for use by the session. Ownership of the connection remains with the client, so don't forget to call `RConnection::Close()` once you have finished using the connection with the HTTP framework. Example 11.1 sets both the `EHttpSocketServ` and `EHttpSocketConnection` session properties – the socket server property to save on resources by sharing the socket server handle between the HTTP framework and the client, and the connection property to allow the client to specify the connection without prompting the user¹.

11.2.2 Transaction Properties

Session properties apply to all transactions within a session. Some properties, such as the use of pipelining or a proxy, can also be set on a particular transaction. These will override the session behavior.

11.3 Creating and Submitting a Transaction

The request and response parts of an HTTP transaction are encapsulated in an `RHTTPTransaction`. Here are the most interesting methods:

```
class RHTTPTransaction
• {
• public:
•   IMPORT_C void SubmitL(THTTPFilterHandle aStart =
THTTPFilterHandle::EClient);
•   IMPORT_C void NotifyNewRequestBodyPartL(THTTPFilterHandle aStart =
THTTPFilterHandle::EClient);
•   IMPORT_C void Cancel(THTTPFilterHandle aStart =
THTTPFilterHandle::EClient);
•   IMPORT_C void Close();
•   IMPORT_C RHTTPRequest Request() const;
•   IMPORT_C RHTTPResponse Response() const;
```

¹ As discussed in Chapter 6, applications may wish to allow the user to configure a connection that should always be used to access the service, rather than prompting each time a connection needs to be created, or using the device-wide default connection.


```

•   IMPORT_C RHTTPSession Session() const;
•   IMPORT_C RHTTPTransactionPropertySet PropertySet() const;
•   IMPORT_C void SendEventL(THTTPEvent aStatus, THTTPEvent::TDirection
aDirection, THTTPFilterHandle aStart);
•   IMPORT_C const CCertificate* ServerCert();
•   IMPORT_C RString CipherSuite();
•   };

```

A *transaction* consists of requests and responses, each of which are encapsulated into a ‘message’ (‘message’ is an abstraction defined in RFC 2616). These requests and responses contain header fields and, optionally, body data.

We now need to create a class derived from `MTransactionCallback` to monitor the progress of this transaction. The interface has two callback functions:

```

class MHTTPTransactionCallback
{
public:
    virtual void MHFRunL(RHTTPTransaction aTransaction, const THTTPEvent&
aEvent)=0;
    virtual TInt MHFRunError(TInt aError, RHTTPTransaction aTransaction,
const THTTPEvent& aEvent)=0;
};

```

These are called by the framework, letting you know, asynchronously via transaction events, how your request is progressing. There are more details on monitoring a transaction in Section 11.5.

HTTP requests are created using `OpenTransactionL()`. You can then use the returned `RHTTPTransaction` to configure the transaction, setting transaction properties via the `RHTTPTransactionPropertySet` returned from `PropertySet()`. You can also modify the request via the `RHTTPRequest` returned from `Request()`. You can add any headers needed in the request, for example `Content-Type`.

For Post requests you need to specify where the body data is obtained from using `SetBody()`. Further details on the use of `SetBody()` are given in section 11.4.

Once you have finished customizing the request, you should submit it by calling `SubmitL()`. If you have not provided an `RConnection` in the HTTP session properties, then submitting the first transaction will trigger the creation of a network connection.

If there is a problem with the request, the transaction will be cancelled. The client application will receive an `ECancel` event via their `MTransactionCallback::MHFRunL()`.

```

EXPORT_C void CFlickrRestAPI::IssueImagePostL(const TDesC& aFileName)
{
    ...

    // Open a POST transaction
    RStringF postMethod;
    RStringPool stringPool = iSession.StringPool();
    postMethod=stringPool.StringF(HTTP::EPOST,RHTTPSession::GetTable());

    iPostTransaction = iSession.OpenTransactionL(uri, *this, postMethod);
    postMethod.Close();

    // We provide the body
    iPostTransaction.Request().SetBody(*this);

    ...

    iPostTransaction.SubmitL();
}

```

Example 11.2 Submitting a Post request

11.4 Supplying Body Data

Example 11.2 shows a Post request being made. The Post method requires a body in the request. To supply this you need to provide an object which implements the `MHTTPDataSupplier` interface.

```

class MHTTPDataSupplier
{
public:
    virtual TBool GetNextDataPart(TPtrC8& aDataPart) = 0;
    virtual void ReleaseData() = 0;
    virtual TInt Reset() = 0;
};

```

For small files you could return all of the data for the body in one response to `MHTTPDataSupplier::GetNextDataPart()`. For larger files you should return the data in parts.

If supplying data in parts it is important to implement `Reset()` and be able to resupply the data if it is called. A Post request may need to be resent if, for example, the network connection is dropped, or the server responds with a redirect. If `Reset()` returns an error the request will fail.

Once `GetNextDataPart()` has been called, the data supplied in `aDataPart` must remain valid, and must not be altered until `ReleaseData()` is called.

11.5 Monitoring a Transaction

11.5.1 Event Handling

The client is notified about the current status of the transaction using transaction events. There are two types of events:

- **Incoming** – generated within the framework and notifying the client (and filters – see section 11.13) of the transaction’s current status.
- **Outgoing** – most of these originate in client actions (e.g. cancelling a transaction will generate an `ECancel` event) and notify the framework of client actions.

For an HTTP session we are only interested in transaction events; however, not all events are relevant. The ones of interest are shown in Table 11.1.

11.5.2 Receiving Header Data

When the event `EGotResponseHeaders` is called it indicates that the status line and response header fields have been received and are accessible. The status code is the most important information. The headers can be retrieved with a call to `RHttpResponse::GetHeaderCollection()`. Through `RHTTPHeaders` individual header values can be retrieved using `GetField()`. Here are the methods provided by `RHTTPHeaders`:

```
class RHTTPHeaders
{
public:
    IMPORT_C TInt FieldPartsL(RStringF aFieldName) const;
    IMPORT_C TInt GetField(RStringF aFieldName, TInt aPartIdx, THTTPHdrVal&
        aHeaderValue) const;
    IMPORT_C TInt GetRawField(RStringF aFieldName, TPtrC8& aRawFieldData)
        const;
    IMPORT_C TInt GetParam(RStringF aFieldName, RStringF aParamName,
        THTTPHdrVal& aReturn, TInt aPartIdx = 0) const;
    IMPORT_C THTTPHdrFieldIter Fields() const;
    IMPORT_C void SetFieldL(RStringF aFieldName, THTTPHdrVal aFieldValue);
    IMPORT_C void SetFieldL(RStringF aFieldName, THTTPHdrVal aFieldValue,
        RStringF aParamName, THTTPHdrVal aParamValue);
    IMPORT_C void SetParamL(RStringF aFieldName, RStringF aParamName,
        THTTPHdrVal aParamValue, TInt aPartIdx);
    IMPORT_C void SetRawFieldL(RStringF aFieldName, const TDesC8&
        aRawFieldData, const TDesC8& aFieldSeparator);
    IMPORT_C TInt RemoveField(RStringF aFieldName);
    IMPORT_C TInt RemoveFieldPart(RStringF aFieldName, TInt aIndex);
    IMPORT_C void RemoveAllFields();
};
```

Table 11.1 HTTP Transaction Event codes

Event Name	Description
EGotResponseHeaders	Indicates that the response has been received and the status line and header field information can be retrieved, for example Check transaction for ContentType or Content-Length information if expecting response body
EGotResponseBodyData	Indicates that body data is ready for access (need to call <code>GetNextBodyPart()</code> ... and then <code>ReleaseData()</code>). It may be the entire message body or a part. If it is the last body part <code>ETrue</code> is returned
EGotResponseTrailerHeaders	Indicates trailing headers have been received and can be retrieved
EResponseComplete	Can be taken as an indication of end of the body (if for some reason last call to <code>GetNextBodyPart()</code> failed to return <code>Etrue</code>)
ESucceeded	Transaction completed OK – just need to close it
EFailed	The client needs to investigate the cause of the failure. It may be a failing status received from the server or a problem with the connection
ERedirectedPermanently	Framework will deal with resending the request to the new location but the client should take note for the future
ERedirectedTemporarily	No need for action. The redirection will be automatic
ERedirectRequiresConfirmation	Client needs to make a decision: either to close the transaction or resubmit it to the new location (already set)

11.5.3 Receiving Body Data

A response message may not be received in one go, especially if the body is large. The framework will indicate that there is body data to retrieve by sending one or more `EGotResponseBodyData` events (see Example 11.3).

```
...
case THTTPEvent::EGotResponseBodyData:
{
    // Part (or all) of response's body data received. Use
    // aTransaction.Response().Body()->GetNextDataPart() to get
    // the actual
    // body data.

    // Get the body data supplier
    MHTTPDataSupplier* body = aTransaction.Response().Body();
    TPtrC8 dataChunk;
    TBool isLast = body->GetNextDataPart(dataChunk);

    if(iCurrentTansactionType == ETransactionGetAuthToken)
    {
        iRestXMLHandler->ProcessGetTokenResponseL(dataChunk,
            iFlickrAuthStruct);
        if(iFlickrAuthStruct)
        {
            StoreAuthorisationDetailsL();
        }
    }
    else
    {
        {
            iTransactionErrorCode = KErrArgument;
        }
    }
    else if(iCurrentTransactionType == ETransactionPostImage)
    {
        TBuf8<50> imageId;
        iRestXMLHandler->ProcessPostImageResponseL(dataChunk,
            imageId);
    }

    // Always remember to release the body data.
    body->ReleaseData();
}
break;
}
...
```

Example 11.3 How to process body data from an HTTP Response

When the event is received, the response body data part is accessible via `GetNextDataPart()`. It is important to call `ReleaseData()` once you have finished processing this part of the body data otherwise you will not receive the rest of the body. If it is the last (or only) part, `GetNextDataPart()` should return `ETrue`; but under some circumstances, especially when the server is using chunked transfer encoding, then the

framework may not be able to reliably signal the end of the body. As is it not possible to control whether the server uses chunked transfer encoding or not, you should also watch for the `EResponseComplete` event, which reliably signals the end of the body data.

11.6 Cancelling a Transaction

At any point after a transaction has been submitted it can be cancelled. The framework may also cancel a transaction, for example when a connection error occurs.

11.7 Closing a Transaction

When the transaction has completed, either with `EFailed` or `ESuccessful`, it should be closed by calling `Close()` on the `RHTTPTransaction`.

`ReleaseData()` must be called prior to closing a transaction. This may appear non-intuitive, however, failing to do so puts the connection in a bad state, and it will not be usable by the subsequent transaction (if there is one queued).

11.8 Stringpool

Stringpool (which is part of the BAFL APIs) is used by the HTTP framework. It has been designed for efficient comparison and storage of standard strings. In the case of HTTP there are a great many strings being used in a session, including the header field names.

Stringpool handles 8-bit strings, either case sensitive and accessed via `RString`; or case insensitive, accessed via `RStringF`. When a session is opened, it creates a stringpool accessible for the lifetime of the session. All commonly used HTTP strings are loaded into the session's stringpool from a static string table. The `RStringPool` API offers the ability to dynamically load new strings and string tables into the string table maintained by the HTTP session. It also offers the ability to translate between the `RStingTokenF` returned from some HTTP APIs and the `RStingF` class.

```
class RStringPool
{
public:
    IMPORT_C void OpenL();
    IMPORT_C void OpenL(const TStringTable& aTable);
```

```

IMPORT_C void OpenL(const TStringTable& aTable,
    MStringPoolCloseCallBack& aCallBack);
IMPORT_C void Close();
IMPORT_C RStringF OpenFStringL(const TDesC8& aString) const;
IMPORT_C RString OpenStringL(const TDesC8& aString) const;
IMPORT_C RString String(RStringToken aString) const;
IMPORT_C RString String(TInt aIndex, const TStringTable& aTable) const;
IMPORT_C RStringF StringF(RStringTokenF aString) const;
IMPORT_C RStringF StringF(TInt aIndex, const TStringTable& aTable) const;
};

```

RString and RStringF are the classes that act as handles to specific strings:

```

class RString : public RStringBase
{
public:
    inline RString Copy();
    inline operator RStringToken() const;
    inline TBool operator==(const RString& aVal) const;
    inline TBool operator!=(const RString& aVal) const;
};

class RStringF : public RStringBase
{
public:
    inline RStringF Copy();
    inline operator RStringTokenF() const;
    inline TBool operator==(const RStringF& aVal) const;
    inline TBool operator!=(const RStringF& aVal) const;
};

```

The major use of the string table in HTTP is to open pre-defined strings from the HTTP protocol to pass them to HTTP APIs. This can be seen in example 11.2 where RStringPool is used to access the HTTP : EPOST string.

All header field names and values are stored as case-insensitive strings. The one exception is cookie values. These are stored as case-sensitive strings.

There are some API oddities that you should be aware of when using Stringpool, where sometimes the APIs do not operate in the standard Symbian manner. Specifically, when creating strings using OpenFStringL(), you need to ensure the object is closed when you have finished with it, as you would expect. For strings already in the stringtable, StringF() is used; in this case you should not close the string.

```

void CFlickrRestAPI::SetHeaderL(RHTTPHeaders aHeaders, TInt aHdrField,
    const TDesC8& aHdrValue )
{
    RStringF valStr = iSession.StringPool().OpenFStringL(aHdrValue);
    RStringF nameStr = iSession.StringPool().StringF(aHdrField,
        RHTTPSession::GetTable())

```

```
CleanupClosePushL(valStr);
THTTPhdrVal val(valStr);
aHeaders.SetFieldL(nameStr, val);
CleanupStack::PopAndDestroy(&valStr);
}
```

Example 11.4 Using Stringpool to access both static and dynamic strings

For example, as shown in Example 11.4, when setting a header field, the header field name would be opened using `StringF()`, the value using `OpenFString()` (unless the value happens to be a commonly used string that is available as a static string, in which case it would also be opened using `StringF()`).

11.9 Proxy Support

Support for setting proxy information is provided at the session level and on a per transaction basis through the use of HTTP properties. When set at the `RHTTPSession` level, the proxy information is used for every transaction, except if overridden by the transaction itself.

```
TStringTable& stringTable = RHTTPSession::GetTable();

RHttpTransaction trans = iSession.OpenTransactionL(...);
RHTTPTransactionPropertySet transInfo = aTransaction.PropertySet();

transInfo.SetPropertyL(
iSession.StringPool().StringF(HTTP::EProxyUsage, stringTable),
iSession.StringPool().StringF(HTTP::EUseProxy, stringTable));

// proxyAddressStr = buffer containing value of proxy address
RStringF proxyAddress =
    iSession.StringPool().OpenFStringL(proxyAddressStr);
CleanupClosePushL(proxyAddress);
transInfo.SetPropertyL(iSess.StringPool().StringF(HTTP::EProxyAddress,
stringTable, proxyAddress);
CleanupStack::PopAndDestroy(&proxyAddress);
```

Example 11.5 Setting a proxy

The property `EProxyUsage` is set to indicate a proxy should be used, the address being specified in the property, `EProxyAddress`. This should include the port number if it differs from the default (8080). Example 11.5 shows how to set proxy information for a specific transaction.

General HTTP proxy information is available in `CommsDat` for each IAP, if that IAP has a proxy set; however, if an application wishes or needs to use its own proxy then it can supply the correct information directly to the framework.

11.10 Cookie Handling

Cookie support is not provided by the framework. Cookies are handled like any other header field, although there are some differences. Cookie field *values* are stored and accessed as *case-sensitive* strings in the stringpool. As the `SetCookie` header can appear multiple times, and the format of a cookie header differs from most other header fields, accessing the data (via `GetField()` and `GetParam()`) is slightly different.

```
TInt cookieCount = headers.FieldPartsL(setCookie);

THTPHdrVal name, value;

RStringF setCookie =
    strP.StringF(HTTP::ESetCookie, RHTTPSession::GetTable());
RStringF cookieName = strP.StringF(HTTP::ECookieName,
    RHTTPSession::GetTable());
RStringF cookieValue = strP.StringF(HTTP::ECookieValue,
    RHTTPSession::GetTable());

// May be a number of set-cookie header fields present in header of a
// response.
TInt ii=cookieCount;
while (--ii)
{
    aHeaders.GetParam(setCookie, cookieName, name, ii);

    aHeaders.GetParam(setCookie, cookieValue, value, ii);

    // Parameters obtained explicitly. Other possible parameters are:
    // EDomain,
    // EMaxAge, ECookiePort, EComment, ECommentURL, ESecure, EDiscard,
    // EVersion,

    RStringF cookieExpires = strP.StringF(HTTP::EExpires,
        RHTTPSession::GetTable());
    if (aHeaders.GetParam(aFieldName, cookieExpires, hdrVal, aPartIndex) ==
        KErrNone)
        // store string cookieExpires.DesC()
}
```

Example 11.6 Cookie handling example

11.11 HTTP Connection Configuration

It is possible to configure the session in order to potentially enhance performance of the framework for a particular application. The session properties can be modified based on the usage patterns that an application exhibits.

11.11.1 Pipelining and Persistent Connections

Pipelining and persistent connections are defined as part of the HTTP/1.1 standard. Persistent connections allow multiple transactions to a server

to be sent over a single socket connection. Pipelining allows one or more new transactions to be submitted to a server over a persistent connection before a previous transaction has completed. The HTTP framework supports both pipelining and persistent connections.

An HTTP client may submit multiple requests to the framework. With a persistent connection (the default in HTTP/1.1), but without pipelining, an HTTP clients can submit multiple transactions in all cases. They are just not queued on the connection. Only one transaction can be active for each socket connection. Multiple connections will be used for requests to different hosts.

With pipelining switched on, multiple transactions to the same host can be sent down one socket without waiting for a previous transaction to complete. Obviously pipelining requires persistent connections in order to operate. Figure 11.2 shows the difference in the request-response pattern for the three types of connection – non-persistent connections, persistent connections without pipelining, and persistent connections with pipelining.

When using network connections with high latency, such as GPRS, the benefits of persistent connections and pipelining become clear – in the case of persistent connections there is no need to pay the cost of setting up and tearing down a TCP connection for each transaction; and in the case of pipelining, there is no need to wait for a server to respond to a previous transaction before sending the next one. For typical web pages, which might consist of a number of small images loaded from the same server, the benefits can be substantial.

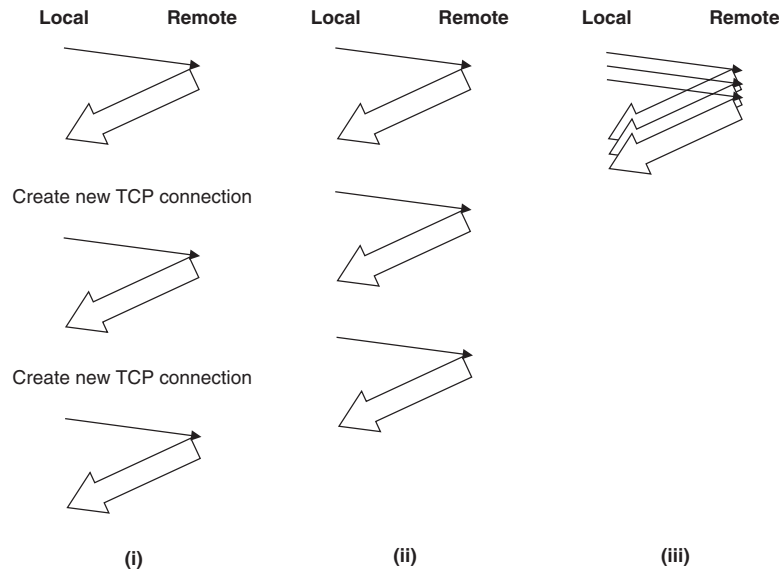


Figure 11.2 Changes to request-response pattern when using persistent connections and pipelining

As an example, let’s do some rough calculations based on network measurements taken in the UK. The round trip time to a server on the Internet over an ADSL connection is about 33 ms. Over a 3G GPRS connection, this rises to 184 ms. Over a 2.5G GPRS connection, the time is 405 ms. Setting up a TCP connection requires three packets before any data can be transferred, but due to the protocol design, the extra cost incurred is actually just over two packets worth of time, or one round trip. Therefore minimizing the number of times we have to bear this cost offers potentially big gains. Imagine we had a web page with 10 small images on it, all from the same server, and that we can use the same persistent connection for all of them². In this case, we’ve saved ourselves 297 ms on the ADSL line, but a massive 1656 ms over 3G and an even bigger 3645 ms over 2.5G.

Pipelining can have similar benefits. Instead of leaving the connection idle in one direction whilst we wait for a response, we can send several requests at once. To calculate the benefits, we need some bandwidth figures for the different technologies. For ADSL, we see about 373 kb/s from servers on the Internet; for 3G GPRS we see about 44 kb/s; and for 2.5G GPRS we see about 6 kb/s.

To work out the benefit of pipelining, we’re going to assume that the request is about 632 bytes. However, we’re going to say that each of those 10 images that are coming back is about 6 kb on average. This gives us the following figures for the total time to download all the images:

As you can see from the figures above, with both persistent connections and pipelining on, it can be possible by correctly setting the following session properties and connecting to a server that supports these features, to achieve a better performance in cases where all requests are independent and to the same server. Obviously as more servers are involved, or as some requests depend on the results of previous ones (as would be

Table 11.2 A rough guide to performance using persistent connections and pipelining (times in ms)

	No persistent connections or pipelining	Persistent connection	Persistent connection and pipelining
2.5G GPRS	19120	15060	10506
3G GPRS	5180	3340	382
‘8 Mb/s’ ADSL	856	516	196

² This isn’t necessarily the case – the server administrator can set an upper bound on how many requests can be made over a single socket connection before the server disconnects, however this number is typically kept quite high (in the hundreds) to improve performance for all clients, not just mobile ones.

the case when downloading the web page itself, rather than the images contained on it) then the performance gains from these features will not be as great. Similarly, in lower latency and higher bandwidth networks the benefits are less pronounced – hence the inclusion of ADSL figures here as an example. WLAN networks should give figures of a similar order of magnitude to ADSL, especially in cases where ADSL is providing the backhaul for the WLAN connection! Certainly in our testing the network latency only went up by a few milliseconds when using WLAN rather than a direct cable connection.

Here is the list of the session properties that can benefit from tuning for your application, and the networks and servers it works with:

- `EMaxNumTransportHandlers` – maximum allowed number of socket connections. Multiple socket connections can be opened either to different hosts, or the same host. It depends on how many transactions have been submitted by the client. By default the maximum number of connections is four. What is optimal depends on the usage of an application, especially the number of servers to which it connects.
- `HTTP::EHttpPipelining` – in an HTTP session, application can send multiple requests without waiting for a response. The HTTP framework has a pipelining property, which by default is on. So multiple requests sent to a particular host will be pipelined down a single socket connection. Note that there are some HTTP server implementations, or particular versions of those implementations, that have problems with pipelining – therefore extensive testing with the servers with which you expect to interact is strongly recommended.
- `HTTP::EMaxNumTransactionsToPipeline` – the maximum number of transactions allowed to be pipelined on a particular connection. By default there is no limit.

11.11.2 Authentication

Both basic and digest authentication are supported. A client application needs to add the authentication filter. To do this the client must implement the interface `MHTTPAuthenticationCallback`:

```
class MHTTPAuthenticationCallback
{
public:
    virtual TBool GetCredentialsL(const TUriC8& aURI, RString aRealm,
RStringF aAuthenticationType, RString& aUsername, RString& aPassword) = 0;
    inline void InstallAuthenticationL(RHTTPSession aSession);
};
```

Calling `InstallAuthenticationL()` will add the authentication filter in the filter queue.

When a 401 response is received from the server the authentication filter will call `GetCredentials()`. The implementation of this should return the username/password information expected – a simple implementation with hard coded name and password is show in Example 11.7.

```
TBool GetCredentialsL(const TUriC8& /*aURI*/, RString aRealm, RStringF
/*aAuthenticationType*/, RString& aUsername, RString& aPassword)
{
    _LIT8(KUsername, "testName");
    _LIT8(KPassword, "changeMe");

    RStringPool stringPool = aRealm.Pool();
    aUsername = stringPool.OpenStringL(KUsername);
    aPassword = stringPool.OpenStringL(KPassword);
    return ETrue;
}
```

Example 11.7 A simple implementation of `GetCredentials()`

11.11.3 Multihoming

As discussed in Chapter 6, Symbian OS supports multiple, simultaneous, independent network connections, which are managed using the `RConnection` API. When opening a socket, it is associated with the `RConnection` that is passed into the `Open()` call. If no connection is passed in when the sockets are opened, the HTTP framework will trigger the creation of the device-wide default network connection, according to the settings in `CommsDat`, upon the submission of the first transaction.

Some client applications may wish to use a network connection other than the default one – for example, they may provide a setting to allow the user to specify that a certain connection should always be used. To get the framework to use this connection, you need to create both an `RSocketServ` and an `RConnection`, then call `RConnection::Start()` with the desired IAP (see Chapter 6 for more details on this). After starting an HTTP session (but before submitting any transactions) the following session properties need to be set:

- `HTTP::EHttpSocketServ` – value is the `RSocketServ` handle
- `HTTP::EHttpSocketConnection` – value is the address of the `RConnection` object

The `RConnection` object must not be a local variable and must be guaranteed to outlive the HTTP session.

Once set, the session will use the supplied objects when opening all the sockets it uses. The client remains the owner of these objects. Therefore the framework will not restart the `RConnection` if it is disconnected. An error code of `KErrNotReady` is a good indication that the network connection has gone down. The client should monitor progress values from the `RConnection` to check for the connection going down, and if so restart the connection and resubmit the failed transaction(s). Again, chapter 6 has more details on the progress notifications feature of `RConnection`.

If the framework is using the default network connection, then the `RSocketServ` and `RConnection` used to open all the sockets are owned by the framework. It will deal with any idle time outs or disconnections that occur on the network connection.

The client can monitor the network connection used by the framework if it so desires. The `RSocketServ` and `RConnection` can be obtained from the connection properties mentioned above. Clients must respect that the framework owns these objects and therefore their lifetime does not extend beyond the closure of the session.

11.12 Platform Security

HTTP is a plug-in framework that runs in the client's thread. Any client using the framework will need the `NetworkServices` capability in order to start network connections for the HTTP stack to use, or to allow the HTTP stack to start a connection automatically when it tries to access the network.

11.13 Filters

Filters are plug-in components which allow for customization of the HTTP framework's behavior. They allow clients to add functionality to the framework in an easy and seamless manner. They are concerned with interactions based upon the type and form of transaction data. The transaction travels in two directions through the chain of filters:

- **Down:** a request sent from the client application, via each filter in turn, to the protocol handler.
- **Up:** a response from the protocol handler, via each filter in turn, to the client application.

Data can be consumed, processed and/or modified by filters anywhere between the client application and the protocol handler. After a session

has been created, the set of filters used cannot be changed during its lifetime.

Once a transaction has been submitted, an outgoing `ESubmit` event will be sent through the framework. Any filters that have registered for this event can then query and/or modify the transaction before it is sent as an HTTP request to the server.

It is useful to have some understanding of filters. There will be some already loaded in the framework and they will affect its behavior. It is useful to check which ones are loaded, as there may be a combination of Symbian-supplied filters and filters from the UI platforms.

A number of core filters are supplied by the framework to perform the following functionality:

- authentication (RFC 2616 Section 14.8)
- redirection (RFC 2616 Section 10.3)
- validation (RFC 2616 Section 13.3).

These are loaded automatically, using `ECom`, when the session is created.

Each `ECom`-based filter can declare itself as one of three types:

- implicitly loading, mandatory – the framework will fail to start up if this filter does not load
- implicitly loading, optional – the framework will try to load these filters, but if they fail the framework startup continues
- explicitly loading – the client must explicitly load these filters into the framework; typically these are filters, such as the authentication filter, than need some interaction with the client.

Filters are declared as one of these three types in the `default_data` section of the `IMPLEMENTATION_INFO` in their `ECOM` registration resource file. The format of the `default_data` section should include the string `'HTTP/ '`, followed by a `'+'` for the mandatory implicitly loading filters; nothing for the optional implicitly loading filters, and a `'- '` for the explicitly loading filters. Then the name of the filter should follow. For example, `'HTTP/+MyVeryImportantFilter'` would force `'MyVeryImportantFilter'` to load, and the framework to fail to initialize if the filter could not be loaded.

At session creation time, any filters that are of an 'implicitly loading' type will be loaded automatically, along with the core filters.

The authentication filter, whilst being part of the core set, is of an 'explicitly loaded' type because it requires a mixin class, `MHTTPAuthenticationCallback`, to be supplied when the filter is created.

This provides a callback interface from the filter to the client, and therefore must be implemented, and therefore supplied explicitly, by the client.

In addition to the core filters there may be UI platform specific filters already loaded into the framework, such as a GZip content encoding filter, or a DRM handling filter. The client can customize and query the list of filters either using `RHTTPFilterCollection`, or the `TFilterConfigurationIterator` to add new, or remove existing, filters.

11.13.1 Writing a Filter

Filters must be derived from `MHTTPFilter` interface:

```
class MHTTPFilter : public MHTTPFilterBase
{
public:
    IMPORT_C virtual void MHFUnload(RHTTPSession aSession, THTTPFilterHandle
        aHandle);
    IMPORT_C virtual void MHFLoad(RHTTPSession aSession, THTTPFilterHandle
        aHandle);
public:
    enum TPositions
    {
        EProtocolHandler = 0,
        ECache = 100,
        EStatusCodeHandler = 200,
        EUAProf = 250,
        ECookies = 300,
        ETidyUp = 400,
        EClientFilters = 500,
        EClient = 1000
    };
};
```

To register, call `AddFilter()` on the `RHTTPFilterCollection` retrieved from the `RHTTPSession` passed in `MHFLoad()`. This will add your filter to the filter queue in a specific position. The position of the filter is specified as a parameter. It must be a number between 0 and 1000 which determines where the filter will be placed between the client at one end and the protocol handler at the other. Care should be taken as position affects the behavior. For example, if two filters act upon the body data which one comes first is important.

Filters can register for the following properties:

1. *Occurrence of an event*: filters can listen for a specific event or all transaction events (`EAnyTransactionEvent`, or use `EAll` to include session events). If a filter is interested in several events, it must register separately for each one.

2. *Presence of a header*: a filter may also be interested in a specific header being present, e.g. `SetCookie`, for a cookie filter.
3. `StatusCode` returned by the server, e.g. the redirection filter registers for the 3xx status codes (300–304 and 307 to be specific).

When the event for which the filter is registered occurs, the filter's `MHFRunL()` will be called. Each filter's `MHFRunL()` is called in the order that the filters are registered. At this point your filter's processing can take place. It works in a similar fashion to `MTransactionCallback::MHFRunL()`. In fact the `MTransactionCallback` implementation can be regarded as the last filter in the queue, which has registered for all incoming events.

11.14 Summary

In this chapter we have learnt about:

- HTTP sessions and transactions, and what features are and are not supported by the HTTP framework
- creating and submitting transactions to the HTTP framework
- sending body data as part of a request, and receiving it as part of a response
- monitoring the state and progress of a transaction
- use of the `Stringpool` to store and access common strings
- how to get the HTTP framework to use a proxy
- how to implement cache and cookie support in the HTTP framework
- how to use filters that need explicit installation into the framework, such as the authentication filter
- options the HTTP framework provides for improving performance, particularly over high-latency connections
- how to create our own filters to modify and extend the behaviour of the HTTP stack.

12

OMA Device Management

12.1 Introduction

This chapter introduces the reader to the Symbian implementation of the Open Mobile Alliance (OMA) Device Management (DM) specification. It is not intended as a comprehensive guide to the specification, but does cover aspects that are relevant to developers who wish to add plug-ins to the Symbian OS DM framework. At the time of writing, the framework plug-in (DM adapter) information contained within this chapter and the companion code is based on a prototype interface. This interface is due to be published in v9.3 of Symbian OS, and although all possible care has been taken to ensure that the information will remain accurate, it is possible that the interface may be subject to change prior to its release. Not all Symbian OS-based devices will include the Symbian DM solution initially; in particular, S60 currently have an independent solution – more details about their implementation, including details for creating Device Management adapters, can be found on the Forum Nokia website.

The DM protocol was originally created by the SyncML Initiative Ltd as a standardized means of managing remote devices. The SyncML Initiative was formed as a not-for-profit corporation to produce an open specification for device management and data synchronization. Prior to the formation of the SyncML Initiative both device management and data synchronization were performed using a variety of proprietary protocols that operated on a limited number of devices and with limited support by remote servers. Due to the nature of these proprietary protocols the support for device management on mobile devices was fragmented and non-interoperable.

In 2002 the SyncML Initiative was integrated into the OMA, and the standardization effort for both device management and data synchronization has continued within the OMA. The OMA standardization effort brings together representatives from a wide variety of companies such

as mobile device manufacturers, server vendors, operators and software companies collaborating on the standard to encompass as many industry aspects as possible.

Device management provides an over-the-air (OTA) method of remotely managing the settings, applications and firmware of a mobile device and accessing diagnostic information. Updating the firmware OTA reduces the time to market by allowing firmware updates to be performed once the end user has the device, rather than in the factory. Support costs for operators are reduced by enabling operators to access settings and diagnostics information on the device during a support call with the end user, without the need to return the device to a support centre. Operators can also update applications on devices or install applications to provide the end user with access to new services as they become available. Finally, enterprises can use device management to manage the large numbers of devices they are providing to their users.

For these reasons, interest in device management has recently started to grow as the advantages are being recognized.

Currently, the OMA device management specification is at revision 1.2, and this is the revision supported by Symbian OS v9.2 onwards. OMA device management v1.1.2 is supported in Symbian OS v8.0 to v9.1. More information on OMA device management and the specification documents can be found on the OMA website, www.openmobilealliance.org.

The following sections of this chapter look at the OMA device management specification, the Symbian frameworks that implement the OMA specification and how to extend the framework by writing a DM adapter to allow DM servers to manage the settings of applications.

12.2 Device Management In Symbian OS

The OMA specification consists of shared components for both device management and data synchronization and therefore the Symbian OS architecture consists of separate frameworks for device management and data synchronization, and a single framework that handles the common aspects of the standard. This can be seen in Figure 12.1.

The SyncML framework handles the connections to remote servers via various transport mechanisms and the encoding/decoding of the data transmissions to the XML-based OMA protocols. The use of the SyncML name is due to the original implementation being created before the OMA took over the standardization effort. The device management and data synchronization frameworks provide APIs which the SyncML framework utilizes to pass commands received in the data transmissions. Once the commands are passed from the SyncML framework to the DM framework, they are queued before being handled. Before the commands are passed

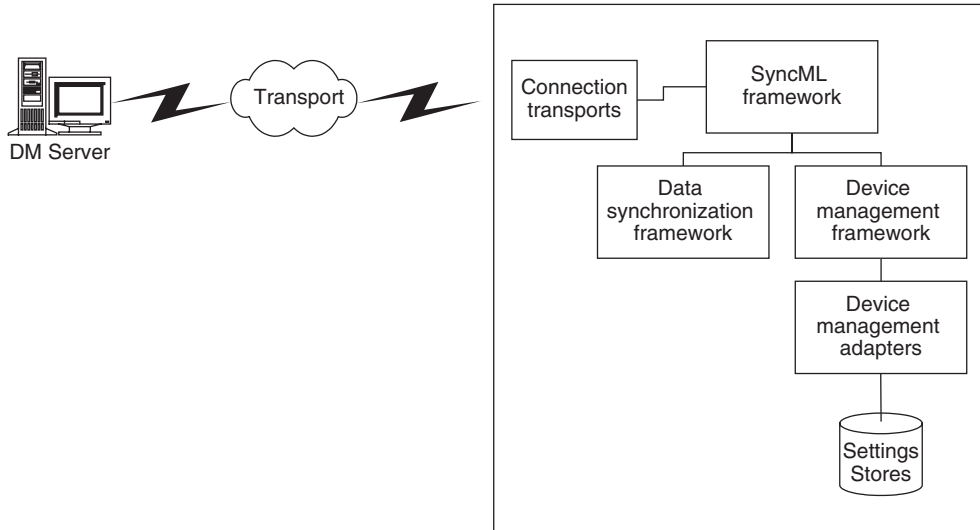


Figure 12.1 A high-level overview of the Symbian OS DM framework

to a specific handler (i.e., a DM adapter), they are first validated to ensure that they can be actioned and that the server requesting the command has the correct access rights for the command.

For maximum flexibility the DM framework utilizes ECOM plug-ins to extend the support for different settings. Plug-ins, referred to as DM adapters, are used to provide an translation layer between the OMA device management-based API, used by the DM framework, and the underlying data store. This allows the DM framework to be extended to support new settings, without changes to the DM framework code itself. To reduce the complexity of the DM adapters, the DM framework also deals with commands on the run-time properties (RTProperties) and description properties (DFProperties) from the DM server (these will be explained in more detail later in the chapter). Commands on RTProperties and DFProperties are not dependant on the underlying data value, and do not affect the data itself, but provide the DM server with information about the setting.

12.3 OMA Device Management Essentials

A DM adapter does not have knowledge of the OMA DM specification, but some knowledge of the specification is essential to implementing an adapter correctly. The following sections of the chapter introduce the reader to elements of the OMA DM specification.

12.3.1 Management Tree

The management tree is used to store settings, which are organized into management objects (more on these later) into a hierarchal tree structure. Settings can be addressed using uniform resource identifiers (URIs). The management tree consists of different types of nodes that form the individual representations of each management object. This allows each setting to be uniquely addressed within the management tree. All management objects within the management tree are addressed from the root node ('/'), and are delimited by '/'. So given the example tree shown in Figure 12.2, to address the node Setting1, the full URI would be '/CommsBook2ndEd/Example/Setting1'.

The management tree store contains information about the known URIs for the DM adapters, and this is utilized when validating and handling property commands from DM servers. The layout of the management tree is determined from the device description framework (DDF) that the DM adapters provide, and from the underlying data that is stored.

12.3.2 Device Description Framework

The DDF holds meta-information about the management tree, the allowable layout of the management tree, and the scope, format, accessibility, etc. of each of the nodes within the management tree. Although the DM framework handles DM commands for DFProperties, the DM adapters must provide the DFProperty information for their management object. The following is a short description of the DFProperties.

Scope – there are two options for the scope of a node: *permanent* or *dynamic*. Permanent nodes are those that are guaranteed to exist in the management tree; dynamic nodes cannot be guaranteed to exist. It is not permissible for a permanent node to be a child of a dynamic node.

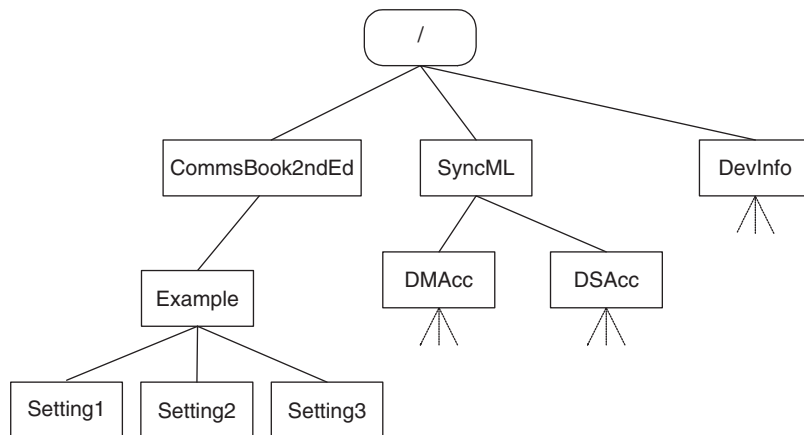


Figure 12.2 An example management tree

DFTitle – this is the human readable name of the node, and is used in the management tree. There are two types of nodes: *interior node* and *leaf node*.

Interior nodes are either nodes that have a fixed name or they can be unnamed. Unnamed nodes do not have a fixed name, but are used to encapsulate common settings that can have multiple entries. The name given to the unnamed node must provide a unique URI to allow its children to be addressable within the management tree.

Leaf nodes have a fixed name, and have data associated with them, this will typically be the value of the setting.

DFFormat – The node format property is the data format of the node value. The node format values are specified by the OMA and include the following:

- b64 – base64 encoded
- bin – binary encoded
- bool – boolean
- chr – text
- int – integer
- node – interior node
- null – null
- xml – XML data.

In the case of interior nodes, the node format value must be 'node'.

DFType – the DFType property is different for interior and leaf nodes.

Leaf nodes – this is the MIME type supported by the leaf node. A leaf node can support more than one MIME type, but they must be registered MIME types.

Interior nodes – for interior nodes the type may be empty, but if a type is given it must only be one registered MIME type.

AccessType – the access type of a node specifies which access commands the node supports. The available access types are:

- Add – add the node
- Copy – copy the node to a new location in the management tree
- Delete – delete the node and any underlying nodes
- Exec – execute the node
- Get – get the node value or children
- Replace – replace the node value or name.

DefaultValue – this specifies the default value of the node when it is created.

Description – a short description of the node where it is felt that more information is useful.

The DDF information can be described by an XML document. The OMA include a framework description to be used for these XML documents in the DTD that is included with the OMA DM specifications.

12.3.3 Management Objects

A management object (MO) is the description of a complete subtree within the management tree. This is typically implemented by a single DM adapter and described by the DDF structure provided by the DM adapter to the DM framework. To describe the MO in a human readable form, the MO is represented by a diagram and a textual description of the DFPProperties for the nodes. The diagram of the MO is similar to the management tree in its layout as it represents the hierarchy of the nodes once they are added to the management tree. For the purposes of DM adapters, the MO must start with the root node and describe all other relationships within the DDF to this node. Each of the interior nodes can have ancestors, a parent and children, whilst the leaf nodes can only have ancestors and a parent. To further describe the layout of the nodes and the possible structures that will be in the management tree, each node must adhere to the following rules:

- Unnamed nodes are represented by a lower case x
- Occurrences are represented as follows:
 - + one or many
 - * zero or more
 - ? zero or one.

If no occurrence character is given then the default occurrence of one must be used.

Figure 12.3 shows the diagram for an MO of the CommsBook2ndEd subtree from Figure 12.2.

In this example, there can be zero or more occurrences of the unnamed node x, but all occurrences must provide a unique URI within the management tree. For example, if a new account was to be created, the DM server could add a new node under ./CommsBook2ndEd called Account1: under this node the leaf nodes Setting1 and Setting2 will always exist but the Setting3 node may or may not exist. Each time a DM server accesses the settings under Account1 it will reference the node using the full URI, so to access Setting1 the DM server would use the URI./CommsBook2ndEd/Account1/Setting1. If the DM server created a second unnamed node Account2, the settings accessed using this node

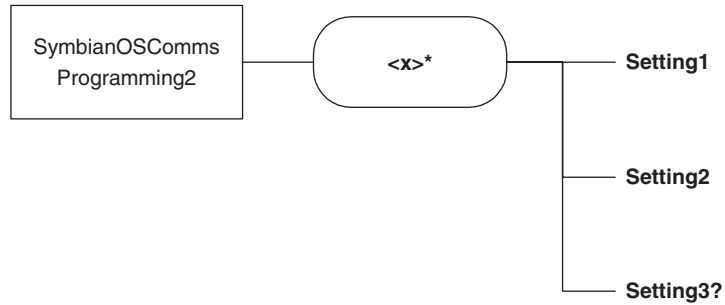


Figure 12.3 An example MO

name in the URI would be stored separately to those of Account1 and therefore are separate to those under the Account1 node. To complete the MO, each node of the MO is described and their DFProperties are listed with the supported values/actions. So the DFProperties for this example might be:

./CommsBook2ndEd

- Scope = permanent
- Occurrence = one
- DFFormat = node
- AccessType = Get
- Default Value =
- Description =

./CommsBook2ndEd/<x>*

- Scope = dynamic
- Occurrence = zero or more
- DFFormat = node
- AccessType = Add, Delete, Get, Replace
- Default Value =
- Description =

./CommsBook2ndEd/<x>*/Setting1

- Scope = dynamic
- Occurrence = one

- DFFormat = chr
- DFTYPE: MIME = text/plain
- AccessType = Get, Add, Replace
- Default Value =
- Description =

./CommsBook2ndEd/ <x>*/Setting2

- Scope = dynamic
- Occurrence = one
- DFFormat = chr
- DFTYPE: MIME = text/plain
- AccessType = Get, Add, Replace
- Default Value =
- Description =

./CommsBook2ndEd/ <x>*/Setting3?

- Scope = dynamic
- Occurrence = zero or one
- DFFormat = chr
- DFTYPE: MIME = text/plain
- AccessType = Get, Add, Replace, Delete
- Default Value =
- Description =

12.4 The Example DM Adapter

The example DM adapter that is included in the code accompanying this book allows a DM server to manipulate the settings of the XMPP application provided as an example for chapter 6. The information provided within this section of the chapter details some of the APIs that are implemented by the DM adapter and by the DM framework. For more information on these APIs and the other APIs available, please see the Symbian OS Library.

The first task when starting a new DM adapter is to determine what settings are going to be needed, and what the occurrences of these settings will be. This is required for the creation of the MO. For the XMPP application it has been identified that there are seven settings that the DM server can manipulate. These are:

- DomainName
- UserName
- Password
- ResourceName
- Language
- ConRef
- UseTLS.

For information on these settings see the example code for chapter 6.

The occurrence for these settings will be one as they always exist when a new account for the XMPP is created. There is one special case here, the ConRef leaf node, which represents a link to a network access point (NAP) within the management tree. In this case the data given is the URI to the NAP in the DM tree. As access points are not DM aware, this URI must be translated to the value that is used to reference the desired access point. This also allows an existing NAP to be used rather than creating new settings for each new XMPP account (or any of the other MOs on the device). The value passed to the node can either be the URI of an existing NAP account, or if the keyword INTERNET is used then the default NAP account must be used. Note that the default NAP account does not have to be explicitly set, as this may be changed at any time by the user.

Finally, as there can be more than one account for the XMPP application, there needs to be an unnamed node to allow the settings to be encapsulated for each account. The occurrence of this unnamed node is zero or more.

The MO for this DM adapter is as shown in Figure 12.4.

12.4.1 DFProperties

./CommsBook2ndEd

- Scope = permanent. This node will always exist in the management tree (whilst the DM adapter is on the device) and cannot be changed, therefore the scope is permanent.
- Occurrence = one. There will only ever be one of these nodes and it must exist.

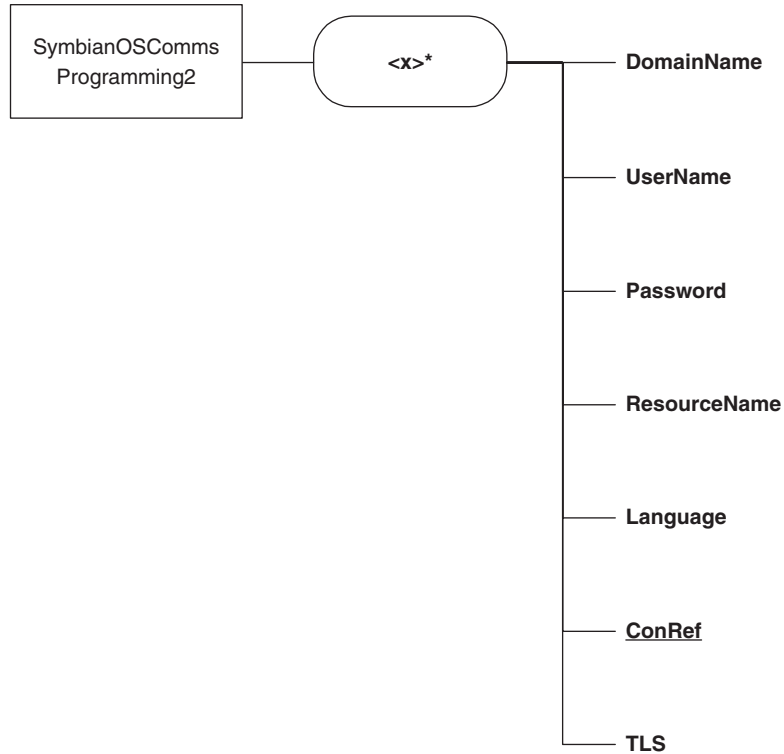


Figure 12.4 The Symbian OS Comms Programming 2 MO

- DFFormat = node. This is an interior node.
- AccessType = Get. It is not possible for permanent nodes to be changed in any way and so Get is the only valid command for this node.
- Default Value =
- Description =

./CommsBook2ndEd/<x>*

- Scope = dynamic. This node will not always exist in the management tree and is therefore dynamic and not permanent.
- Occurrence = zero or more. It is possible for there to be no accounts for the XMPP application or multiple accounts and so this node has an occurrence of zero or more.
- DFFormat = node
- AccessType = Add, Delete, Get, Replace. This node can be added to create a new account, or deleted to delete an entire account. A Get on

this node will return a list of its child nodes. The Replace allows the DM server to change the unique name of an unnamed node – this will be performed by the DM framework and does not require interaction with the DM adapter.

- Default Value =
- Description =

./CommsBook2ndEd/<x>*/DomainName

- Scope = dynamic
- Occurrence = one
- DFFormat = chr. This node is a leaf node that has a text value.
- DFTYPE: MIME = text/plain. This node supports the text/plain MIME type.
- AccessType = Get, Add, Replace. Add is supported to allow a new unnamed node and all child nodes to be added in a single transaction.
- Default Value =
- Description =

./CommsBook2ndEd/<x>*/UserName

- Scope = dynamic
- Occurrence = one
- DFFormat = chr
- DFTYPE: MIME = text/plain
- AccessType = Get, Add, Replace
- Default Value =
- Description =

./CommsBook2ndEd/<x>*/Password

- Scope = dynamic
- Occurrence = one
- DFFormat = chr
- DFTYPE: MIME = text/plain
- AccessType = Add, Replace. Only Add and Replace are supported as the reading back of the password is seen as a security risk.

- Default Value =
- Description =

./CommsBook2ndEd/<x>*/ResourceName

- Scope = dynamic
- Occurrence = one
- DFFormat = chr
- DFTYPE: MIME = text/plain
- AccessType = Get, Add, Replace
- Default Value =
- Description =

./CommsBook2ndEd/<x>*/Language

- Scope = dynamic
- Occurrence = one
- DFFormat = chr
- DFTYPE: MIME = text/plain
- AccessType = Get, Add, Replace
- Default Value =
- Description =

./CommsBook2ndEd/<x>*/ConRef

- Scope = dynamic
- Occurrence = one
- DFFormat = chr
- DFTYPE: MIME = text/plain
- AccessType = Get, Add, Replace
- Default Value =
- Description =

./CommsBook2ndEd/<x>*/UseTLS

- Scope = dynamic

- Occurrence = one
- DFFormat = bool. This leaf node has a Boolean value.
- DFType: MIME = text/plain. This is text/plain as the leaf node expects a text representation for the Boolean values.
- AccessType = Get, Add, Replace
- Default Value = FALSE. If this leaf node is not explicitly added then the default value will be false.
- Description = If 'TRUE' then a TLS connection is used. Expected values TRUE or FALSE.

This MO information can be published to allow operators of DM servers to determine how to setup and change the accounts for the example XMPP application from this book.

The next task is to identify the operations we need to perform using the DM adapter.

Operations to perform on interior nodes:

- Adding an unnamed node – when an Add command is received on the unnamed node, this signifies that a new account is to be added and that the account is associated with the unique node name contained in the URI.
- Deleting an unnamed node – when a Delete command is received on an unnamed node, the account associated with that unique node name must be removed.
- Get on the CommsBook2ndEd node – when a Get command is received on the CommsBook2ndEd node, the DM adapter must return a list of the unique node names for all of the accounts. There are two considerations here:

–accounts are already in the management tree – methods are available from the DM framework for mapping the account IDs to the unique node names.

–accounts that are not in the management tree – the DM adapter must generate new unique node names for all accounts that have no mapping to the management tree.

- Get on an unnamed node – when a Get command is received on an unnamed node, the DM adapter must return a list of all the child nodes of that node (i.e., the settings that exist for that account), which exist on the device.

Operations to perform on leaf nodes (for all of these nodes the account to be accessed is specified by the unnamed node in the URI):

- Adding or replacing a ConRef leaf node – when an Add or Replace command is received for the ConRef leaf node, the DM adapter must convert the value, given as URI in the management tree, to a table entry in the CommsDat database. The table entry value is then stored in the account data. If the keyword INTERNET is given as the URI, then the account data is stored as zero.
- Adding or replacing a UseTLS leaf node – when an Add or Replace command is received for the UseTLS leaf node, the DM adapter must convert the text/plain representation of the Boolean value to the correct logical value and store this value in the account data.
- Adding or replacing all other leaf nodes – when an Add or Replace command is received for a leaf node, the associated settings data must be changed to the given value.
- Get on the ConRef leaf node – when a Get command is received for the ConRef leaf node, the DM adapter must convert the CommsDat table entry from the value stored in the account data into a URI in the management tree. This URI will reference the access point that owns the settings for that CommsDat table. If the account data is zero, then the keyword INTERNET is returned to show that the default access point is used.
- Get on the UseTLS leaf node – when a Get command is received for the UseTLS leaf node, the DM adapter must convert the logical value from the account data to the text/plain representation.
- Get on the Password leaf node – a Get command for the Password leaf node should never be received as the DDF specifies that the node does not support the Get access type.
- Get on all other leaf nodes – when a Get command is received for a leaf node, the DM adapter must return the value stored in the account data.

Other design considerations:

- Atomic commands – the DM adapter must be able to handle multiple commands in a single atomic transaction as described in the OMA DM specification. Transactions can span multiple DM adapters and if any of the commands fail then the entire transaction is rolled back. There are two parts to a transaction, the start and end of the transaction. The start signifies that all following commands (prior to the end notification) are within the transaction. The end of the transaction is either the rolling back of the commands within the transaction or the committal of the transaction.

- Rolling back atomics – when requested, the DM adapter should attempt to revert the changes made within the transaction. The DM adapter must report the success or failure of the reversion for each of the previously completed commands.
- Committing atomics – when requested, the DM adapter must commit the changes that were made within the transaction to ensure that they are persistent.
- Handling commands on multiple accounts – when necessary the DM adapter must be able to handle commands on multiple accounts. This is particularly relevant if multiple accounts are added, changed or both are within a single atomic transaction.

Let's now take a look at the code that implements the DM adapter we've just described. The first tasks to perform when implementing a DM adapter are to create the resource file that will register the DM adapter as an ECOM plug-in to the DM framework and the project MMP file. To register it as an ECOM plug-in to the DM framework the DM adapter must have a resource file in which it declares the interface `_uid` as 0x102018B4. The DM adapter must also state within the MMP file that the `TARGETTYPE` is `PLUGIN`: this will create the appropriate binary for an ECOM plug-in.

In order for the DM framework to be able to communicate with the DM adapters, each DM adapter must implement the specified ECOM interface detailed in `<devman/SmlDmAdapter.h>`. This provides the interface the DM framework requires to access the DDF structure of the DM adapter, pass commands from DM servers and to notify the DM adapter of atomic operations. The classes defined in this file are shown in Table 12.1.

For the DM framework to be able to utilize the DM adapter, the DM adapter must derive from the `CSmlDmAdapter` class; this requires the DM adapter to implement methods to:

- provide the DDF structure the DM adapter supports
- provide a version number for the DDF information
- update leaf nodes (these do not differentiate between Add and Replace commands)
- delete a node and its children
- get leaf node data and the size of the data
- get the children of an interior node
- add an interior node
- perform an Execution on a node

- copy node(s) to another location
- handle atomic transactions
- stream large data
- complete any remaining commands.

Table 12.1 Defined classes

Class name	Description
<code>MSm1DmAdapter</code>	Abstract interface containing the pure virtual methods for <code>CSm1DmAdapter</code> . Developers may find this useful for testing, but should not use it as a direct base for their own adapters
<code>CSm1DmAdapter</code>	The ECOM interface to be derived from by all DM adapter plug-ins
<code>TSm1DmMappingInfo</code>	Encapsulation of a mapping between a node of the DM tree and some external settings store
<code>MSm1DmCallback</code>	Services provided by the DM framework to DM adapter plug-ins, including access to mapping data, access to other nodes of the DM tree, and callbacks for completion of DM commands
<code>MSm1DmDDFObject</code>	Representation of single node in the adapter's DDF
<code>TSm1DmAccessTypes</code>	The set of DM commands that can be permitted on a particular node in the adapter's DDF

To be able to access methods that are provided by the DM framework for use by DM adapters, the `CSm1DmAdapter` class provides a method `Callback()` that returns a reference to an object of type `MSm1DmCallback`. This object can then be used to call selected methods of the DM framework.

All of the DM command interface methods provide the URI to identify the node that the command is for, but this does not translate as easily to C++. To determine which node the command is for, it is necessary to split the URI on the '/' separator. This will provide two useful pieces of information: the depth or number of nodes in the URI and the name of the node the command is for. The number can be used as an easy way to determine if the command is being performed on the DM adapter parent

node, the unnamed node or a leaf node. The name of the node can then be used to identify which leaf node the command is for. Is it important to remember when determining the number of nodes in the URI, that the root node '.' is stripped by the DM framework as it is not used within DM adapters.

The DM framework uses a call to the interface method `DDFStructureL()` to retrieve the DDF information for the DM adapter. The DM adapter builds its DDF structure by calling methods on the passed reference to an `MSm1DmDdfObject` object. Each node of the MO is represented by an object of type `MSm1DmDdfObject` and the reference provided by the DM framework is for the root node. There are two methods provided for adding child nodes, one is to add named nodes and the other to add unnamed nodes. These are respectively:

```
MSm1DmDdfObject& AddChildObjectL(const TDesC8& aNodeName)
MSm1DmDdfObject& AddChildObjectGroupL()
```

Both methods return the reference to a `MSm1DmDdfObject` object that represents the new node. This reference can be used to add the DDFProperty information of the node and also to create the children of the node.

```
void SetAccessTypesL(TSmlDmAccessTypes aAccessTypes)
void SetDefaultValueL(const TDesC8& aDefaultValue)
void SetDescriptionL(const TDesC8& aDescription)
void SetDFFormatL(TDFFormat aFormat)
void SetOccurrenceL(TOccurrence aOccurrence)
void SetScopeL(TScope aScope)
void SetDFTitleL(const TDesC8& aTitle)
void AddDFTypeMimeTypeL(const TDesC8& aMimeType)
```

These references to `MSm1DmDdfObject` objects are only valid for the duration of the call to the `DDFStructureL()` method and the DM adapter must not retain pointers to them. The format of the DDF version is not defined by the DM framework or the OMA and can be an empty descriptor.

To assist the DM adapter in searching for the location of account data, the DM framework provides methods to set and retrieve local unique identifiers (LUIDs) for each node of the management tree. These LUIDs can be used to provide mapping information between the management tree URI and the account to which they refer. For each of the interface methods that handle DM commands, the DM framework will pass the LUID information for the URI. The LUIDs are available via a hierarchical search, so even if the node has no LUID associated with it, the provided LUID may be from its parent or an ancestor. The methods for setting and retrieving LUIDs are respectively:

```
void SetMappingL(const TDescC8& aUri, const TDescC8& aLuid)
HBufC8* GetLuidAllocL(const TDescC8& aUri)
```

For the example adapter, the ID of the XMPP account is used as the LUID for an unnamed node. As the LUIDs are hierarchical, the interface methods to handle DM commands on the children of the unnamed node will also be supplied the correct LUID.

To indicate to the DM framework the status of a DM command, the DM command interface method includes an argument that has a status reference for the command. This status reference is used by the DM adapter to indicate either a successful command handle or an error value. The permissible values for the returned status are defined by the enumerated type `MSmLmAdapter::TError` and are as shown in Table 12.2.

Table 12.2 Permissible values for returned status

Status code	Description of appropriate use
EOK	When the command has been successful
ENotFound	If the URI supplied doesn't correspond to a valid setting in the setting store, or the LUID supplied with the command references a setting (or group of settings) that is no longer valid in the mapped settings store
EInvalidObject	If the value expected in an <code>UpdateLeafObjectL</code> should belong to a set of enumerations but wasn't a valid member, wasn't recognized, or isn't supported
EAlreadyExists	If the URI supplied corresponds to a setting that is already in the setting store
ETooLargeObject	If the object supplied to <code>UpdateLeafObjectL</code> or <code>ExecuteCommandL</code> is too large for the setting mapped to by a URI
EDiskFull	If during processing of the command an out-of-disk or out-of-memory condition occurred
EError	A general error code that can be used if no others apply
ERollbackFailed	Used to complete outstanding commands, if the <code>RollbackAtomicL</code> call is used but the command can't be reverted

Table 12.2 (*continued*)

Status code	Description of appropriate use
EObjectInUse	Used to indicate that the setting can't be modified, accessed or deleted because it is in use by another process
ENoMemory	Used to indicate that memory allocation failed during processing of a command
ERollbackOk	Used to indicate that the roll-back operation was successful
ECommitOk	Used to indicate that the commit operation was successful
ECommitFailed	Used to indicate that the commit operation failed

The status of a command is reported to the DM framework via the callback method:

```
void SetStatusL(TInt aStatusRef, MSmlDmAdapter::TError aErrorCode)
```

Standard Symbian OS error values must not be used for reporting the status of a command. The setting of the status value is also used to indicate to the DM framework that the DM command has been fully handled and all result data has been returned.

In a similar way to the status reference, each of the Get DM command methods provide the argument `aResultsRef`, which allows the DM adapter to specify which DM command the result data is for. The result data is returned via two callback methods:

```
void SetResultsL(TInt aResultsRef, CBufBase& aObject, const TDesC8& aType)
void SetResultsL(TInt aResultsRef, RReadStream*& aStream, const TDesC8&
    aType)
```

The first method is used to return data that is below the streaming size of the DM adapter. The second method is used for returning large objects as an `RReadStream` object, this allows the DM framework to read the data in chunks to reduce the memory usage and the DM framework is responsible for closing the `RReadStream` when it has finished reading the data.

The interface methods for DM commands do not exactly match those specified as part of the OMA DM specification.

For Get DM commands, there are three methods:

```
void FetchLeafObjectL(const TDesC8& aURI, const TDesC8& aLUID, const
    TDesC8& aType, TInt aResultsRef, TInt aStatusRef)
void FetchLeafObjectSizeL(const TDesC8& aURI, const TDesC8& aLUID, const
    TDesC8& aType, TInt aResultsRef, TInt aStatusRef)
void ChildURLListL(const TDesC8& aURI, const TDesC8& aLUID, const
    CArrayFix<TSmlDmMappingInfo>& aPreviousURISegmentList, TInt
    aResultsRef, TInt aStatusRef)
```

The first two methods are specific to leaf nodes only, `FetchLeafObjectL()` is used to retrieve leaf node values, and `FetchLeafObjectSizeL()` is used to retrieve the size of the data that would be returned for a `FetchLeafObjectL()` call. The third method, `ChildURLListL()`, is specific to interior nodes and the DM adapter must return a list of the children of the interior node – each child node in the list is separated by a `‘/’`. The `CArrayFix<TSmlMappingInfo>` argument is provided by the DM framework to assist in listing the children of an unnamed interior node – the array contains a list of the children of the node and their associated LUIDs. Here, the first action is to get a list of all the account IDs and use these to compare against the passed array values; this will provide a list of known child node names. If there are accounts that are added through routes other than the DM adapter, e.g., manually by the user through a configuration user interface in the application, then these also need to be added to the list, but a new unique name will need to be generated for them. Any generated name should be compared to the passed list to ensure that they are unique, and once the uniqueness of the generated name is confirmed, then the LUID for this new URI must be set. The easiest method of generating unique names is to use a name and append an integer value to the end.

For Add and Replace DM commands there are three methods:

```
void UpdateLeafObjectL(const TDesC8& aURI, const TDesC8& aLUID, const
    TDesC8& aObject, const TDesC8& aType, TInt aStatusRef)
void UpdateLeafObjectL(const TDesC8& aURI, const TDesC8& aLUID,
    RWriteStream*& aStream, const TDesC8& aType, TInt aStatusRef)
void AddNodeObjectL(const TDesC8& aURI, const TDesC8& aParentLUID, TInt
    aStatusRef)
```

The first two methods are specific to leaf nodes, and encompass both the Add and Replace DM commands. The third method is to add interior nodes – both named and unnamed ones. The second method is provided to support large objects – if the data being passed by the DM server is greater than the streaming size specified by the DM Adapter, then this method will be called.

For Delete DM commands there is one method:

```
void DeleteObjectL(const TDesC8& aURI, const TDesC8& aLUID, TInt
    aStatusRef)
```

For Copy DM commands there is one method:

```
void CopyCommandL(const TDesC8& aTargetURI, const TDesC8& aTargetLUID,
    const TDesC8& aSourceURI, const TDesC8& aSourceLUID, const TDesC8&
    aType, TInt aStatusRef)
```

For Execute DM commands there are two methods:

```
void ExecuteCommandL(const TDesC8& aURI, const TDesC8& aLUID, const
    TDesC8& aArgument, const TDesC8& aType, TInt aStatusRef)
void ExecuteCommandL(const TDesC8& aURI, const TDesC8& aLUID,
    RWriteStream*& aStream, const TDesC8& aType, TInt aStatusRef)
```

In the methods that the DM adapter implements to handle DM commands, there are two that pass an argument of type `RWriteStream`. There are two further methods that the DM adapter must implement that relate to streaming large objects:

```
TBool StreamingSupport(TInt& aItemSize)
void StreamCommittedL(RWriteStream& aStream)
```

`StreamingSupport()` is used to determine if DM adapters support the streaming interfaces and the size above which the DM adapter expects the streaming interfaces to be used. `StreamCommittedL()` is used by the DM framework to indicate to the DM adapter that it has read all the data from the `RWriteStream` object.

There are three interface methods specific to atomic transactions:

```
void StartAtomicL()
void CommitAtomicL()
void RollbackAtomicL()
```

The DM framework will call `StartAtomicL()` to signal the start of a new transaction and all commands received before this method is called must be handled before this method completes. The DM framework will call `CommitAtomicL()` to signify the successful end of the transaction, all commands within the transaction should be persisted before this method completes. The DM framework will call `RollbackAtomicL()`

to signal the end of the transaction and to request that the DM adapter rolls back all commands handled within the transaction. The DM adapter must, within `RollbackAtomicL()`, return a new status for each successfully handled command indicating if the command was rolled back or not.

There is one unique interface method that is used by the DM framework to signal to the DM adapter that there are no more commands within the transaction, or within the current DM session:

```
void CompleteOutstandingCmdsL()
```

When this method is called, the DM adapter must handle any remaining commands that it has been sent. This is specific to commands that are deferred for later handling, this typically occurs when the DM adapter is not able to action the DM command without information provided in other DM commands. It is not recommended that DM commands are deferred by DM adapters unless it is necessary – in the case of most DM adapters deferring commands will not be necessary.

The handling of DM commands on the ConRef link node is a special case and is worth considering separately. The implementation presented here assumes that the access points are stored under the URI `'./NAP'` in the management tree, and that the LUID for each of the unnamed nodes is the ID in the CommsDat database for the access point. It is highly recommended that any DM adapter that utilizes a ConRef link style node verifies this assumption to prove the solution will work on all the targeted devices. Translating from the passed ConRef value to an access point ID is achieved by using the passed value as the URI for a call to `GetLuidAllocL()`: the returned LUID is stored as the account data. If the passed value is the keyword `INTERNET`, then the account data is set to zero. Translating from the access table number to the management tree URI requires a lookup of all the children to the `'./NAP'` node and finding the one that has an LUID that matches the value stored in the account data. A call to the following method will provide an array of child node names and their LUIDs for the passed URI:

```
void GetMappingInfoListL(const TDesC8& aURI, CArrayFix<TSmlDmMappingInfo>&
    aSegmentList)
```

A search through the array will determine if the stored value has an access point in the management tree. The child node name in the `TSmlDmMappingInfo` object does not include the full URI, and so the node name must be prepended by the rest of the URI prior to the result being returned. If the account data is zero, then the keyword `INTERNET` is returned.

Most DM adapters that only manage settings, such as our example XMPP DM adapter, do not need to support the full set of DM Commands.

Therefore, the Copy and Execute interfaces are not supported to reduce the complexity of the DM adapter, and the data for the XMPP settings is too small to need the support of the streaming interfaces. However, these interface methods must be implemented by the DM adapter regardless of whether they are supported. A suitable error message should be used to show that the command failed.

As is always the case, the DM adapter will require testing. Unit tests will prove that the DM adapter code is sound, does not leak memory and that all the code can be reached, but does not prove that the DM adapter can be used by a DM server to access the settings data. There are three ways to test with a DM server: the first is to use the SyncML Conformance Test Suite (SCTS) provided by the OMA (available as open source from *sourceforge.net*), search for freely available DM servers, or one of the commercially available DM servers.

12.4.2 OMA-specified Management Objects

The OMA has started to standardize management objects to allow for greater compatibility between devices. The DevInfo, DevDetail and Device Management Account management objects are specified as part of the OMA DM specification, but work has begun on a number of other management objects, including connectivity MOs, a software component MO and a firmware update MO. A complete list and information on these MOs can be found on the OMA website.

12.5 Summary

In this chapter we have learnt how about:

- the basic principles behind device management
- how to describe our manageable settings in terms of DFProperties, and the attributes of those settings
- how to implement a DM adapter to manage settings for our application.

Section IV

Development Tips

13

Setting Up for Development

In this chapter we describe a number of ways of attaching hardware to the emulator to enable developers to use the technologies listed in Section II of this book. We will describe methods for attaching Bluetooth hardware, IrDA hardware or telephony hardware to the emulator, and a number of ways of connecting the emulator to an IP network.

Table 13.1 shows the technologies from Section III, and which of the bearer technologies below you can use for testing. Obviously in some cases you'll need extra infrastructure for testing, for example OMA DM needs a device management server. To prevent this chapter becoming unmanageably long, and to keep the focus on Symbian OS, we won't cover setting up servers for testing here – there's plenty of information already on the Internet about that topic.

Important note applying to all sections: when configuring serial ports on the emulator, note that Windows and Symbian OS use different numbering schemes – COM1 on Windows is COMM::0 on Symbian OS, COM2 is COMM::1, etc.

13.1 Bluetooth

There are three main ways of connecting Bluetooth hardware to the Symbian emulator – one uses USB and the other two use a serial port.

At this point we should mention that when using a serial port, there are two protocols that can be used to communicate between Symbian OS and the Bluetooth hardware – called BCSP and H4. The hardware will support one or the other, so you need to know which protocol is supported to be able to configure it.

Table 13.1 Technologies from Part 3

Technology	Bearer technology
Messaging	<p>To use POP3, IMAP4 and SMTP, follow the steps in the 'Network connections for IP' section.</p> <p>Follow the steps in the 'Telephony' section to allow reception of SMS messages.</p> <p>To test reception of SMS and MMS messages on the S60 emulator, you can also use the functionality contained in Tools->Utilities->Events->Messaging events to insert fake SMS and MMS messages directly into the messaging store.</p>
SendAs	<p>To use SendAs over Bluetooth, follow the steps in the 'Bluetooth' section.</p> <p>To use SendAs over infrared, follow the steps in the 'IrDA' section.</p> <p>To use SendAs over SMS, follow the steps in the 'Telephony' section.</p> <p>To use SendAs over MMS, follow the 'Using a serial link to a mobile phone' method in the 'Network connections for IP' section.</p> <p>To test the SendAs 'to flickr' service, follow the steps in the 'Network connections for IP' section.</p>
OBEX	<p>To use OBEX over Bluetooth, follow the steps in the 'Bluetooth' section.</p> <p>To use OBEX over infrared, follow the steps in the 'IrDA' section.</p>
HTTP	Follow the steps in the 'Network connections for IP' section.
OMA Device Management	Follow the steps in the 'Network connections for IP' section.

13.1.1 Setting up the Easy Way – Using the SDK Configuration Dialogs

In the S60 3rd edition emulator, go to the 'Tools' menu and choose 'Preferences', then select the 'PAN' tab, enable Bluetooth and choose the appropriate HCI type and port number.

For the UIQ3.0 SDK, launch the SDKConfig app, go to the 'Communications' tab and select the appropriate COM port. If you need to change the HCI in use, you'll need to edit the configuration file manually.

13.1.2 Setting up the Manual Way – Editing the Configuration File

Using Bluetooth hardware with a serial connection

There are two options for connecting Bluetooth hardware using a serial port – one uses a traditional RS-232 port, the other uses Bluetooth hardware packaged as a compact flash (CF) or PC card.

The first option is to use the same hardware as the Symbian Bluetooth team use for testing and development: Casira pods from Cambridge Silicon Radio (CSR). The Casira pods have a replaceable CSR BlueCore module inside, and both serial and USB interfaces on the back. Since connecting hardware to the emulator via USB is far more difficult than using a simple serial port, the supported option is the serial port.

The second option is far better if you travel a lot and need a portable solution. It is possible to use a CF card-based Bluetooth module, such as the Brainboxes BL-565. When you insert this card in your laptop, Windows will ask you which driver to install. Use the advanced install option to select the driver yourself, prevent Windows from searching for a driver, and tell it that you will select the driver manually. The type of driver you want is 'Ports (COM & LPT)', then in '(Standard port types)' you can select 'Communications Port'. Windows warns you that the driver may not be compatible with your hardware – just say 'Yes' to continue installation.

Once you've installed the driver, go to Device Manager in Windows to find out which COM port number your card was assigned. Make a note of this, and then use either the SDK configuration dialogs, or the manual method described next, to configure the correct port number in Symbian OS. The Brainboxes card we used supported the BCSP protocol, so make sure you select the correct HCI type when configuring the emulator. Once configured, the hardware should just work.

Whichever hardware you are using, you must set the appropriate COM port number in the `bt.bt.esk` file, which on the emulator can be found in `/epoc32/release/winscw/udeb/z/private/101f7989/esock/`. The `[hci]` section contains a `port=` attribute, which should be set to the appropriate Symbian OS COMM port number, bearing in mind the note earlier in this chapter about the difference between Symbian OS and Windows COM port numbering!

Make sure that you first type `port=` and then the number. There must be no space between the `port` and the equals sign. Then insert

exactly one space, then the port number - otherwise the S60 SDK tools may not be able to parse and configure the file.

In previous releases of Symbian OS (v7.0 to v8.0), the file was called `bt.esk` and lived in `/epoc32/winscw/c/system/data/`. In v8.1 the file was renamed to `bt.bt.esk`, but remained in the same location.

In very early releases (pre-v7.0, and some versions of 7.0s and 8.0a such as the ones shipped in the S60 2nd edition SDKs), it lived in `/epoc32/wins/c/system/data/`.

The appropriate HCI can be chosen by altering the `hcidllfilename=` tag to either `hci_bcsp.dll` or `hci_h4.dll`.

Using Bluetooth hardware with a USB connection

This is a new feature in the S60 3rd edition, Feature Pack 2 SDK; although the configuration option to use a USB HCI appears in the Feature Pack 1 SDK, but has no effect. However, there is a download available from the Forum Nokia website (<http://www.forum.nokia.com>), called 'S60 SDK BT driver', that can be used with the S60 3rd edition SDK, and the S60 3rd edition, Feature Pack 1 SDK. After it has been installed on the 3rd edition, Feature Pack 1 SDK, the menu option to use a USB HCI works correctly. Instructions are included in the package for installing the driver and configuring the emulator to use it. We used a TDK USB Bluetooth dongle to test this functionality, which worked successfully with in conjunction with S60 3rd edition, Feature Pack 1 SDK.

13.2 IrDA

Infrared is supported on the emulator with the addition of an IR pod. As with Bluetooth, IR pods are connected via RS-232 to the emulator. The Symbian OS emulator does not support IR pods that connect via USB. Using USB-to-serial adaptors to connect serial-based IR pods is also not recommended, as the speed at which the IR pod operates is typically set by wiggling the flow control lines, and many USB-to-serial adaptors do not perform this operation faithfully – leaving Symbian OS and the IR pod disagreeing about which state they are in.

There are 10 different IR pods supported, although experience shows that even IR pods that are not supported are likely to conform to the

signalling used by a supported pod. The best advice is to experiment and see!

The supported pods are:

- Actisys 220L
- Actisys 220L+
- Actisys 220Li
- iFoundry 8001a

The following models are not as well supported, are supported but have been discontinued, or are harder to find:

- Tekram IR-210
- ConnectTech (unspecified model)
- Extended Systems Jeteye 7201
- Extended Systems Jeteye 7401 (also supports the 7501) – often labelled XTNDAccess instead of Jeteye
- Parallax (unspecified model).

In all cases, Symbian OS needs a basic serial port interface to the IR pod. This means you must not install any Windows drivers that come with the IR pod you are using. You can see if there are any IR drivers installed by looking in the Windows Device Manager and checking for 'Infrared devices'. If you see your IR pod listed there you will need to uninstall the Windows drivers before Symbian OS can use it.

13.2.1 Setting up the Easy Way – Using the SDK Configuration Dialogs

The easiest way to choose the port number in the UIQ3.0 SDK, using the SDKConfig app, is to go to the 'communications' tab and to select the appropriate port. However, if you want to use a pod that isn't compatible with JetEye 7401, such as the Jeteye 7501, Parallax or iFoundry 8001a, then you'll have to edit the configuration file as indicated later.

Similarly, if you want to alter the settings for the S60 3rd edition SDK, you'll need to start the emulator, go to the 'Tools' menu and choose 'Preferences', then select the 'PAN' tab, enable IrDA and choose the appropriate COM port. Again, if you need to use a pod other than the default Jeteye 7401, you'll need to edit the config file manually.

Note that these user interfaces for the selection of the COM port use the Windows numbering system, not the Symbian OS one!

13.2.2 Setting up the Manual Way – Editing the Config File

The config file you need to edit is called `ir.irda.esk`, and is located in the `/epoc32/release/winscw/udeb/z/private/101f7989/esock/` directory.

The two settings you need to change are the `irPod` and `irPhysicalCommPort`. The `irPhysicalCommPort` should be set to the value of the Symbian OS COMM port to which the IR pod is attached, bearing in mind the note at the beginning of this chapter about the differences in numbering between Symbian OS and Windows COMM ports.

The `irPod` value should be set to one of following nine options, depending on which IR pod you are using:

`actisys2201`, `actisys2201+`, `actisys2201i`, `tekram`, `jeteye7201`, `jeteye7401`,
`parallax`, `ifoundry8001a`, or `connectTech`

As per the Bluetooth case, make sure the '=' sign on the config is immediately after the variable name, then there is precisely one space, and then the value, for example `irPod= actisys2201`.

13.3 Network Connections for IP

There are various different ways to connect the emulator to an IP network – which one you choose depends on a combination of personal preference, what hardware you have available, whether you need access to a specific network (e.g., for sending and receiving MMS) and which you can get to work for your particular setup.

Table 13.2 provides a list of these options (which we'll refer to by a short name) and the situations in which you can use them.

The first three methods of connection are the preferred methods – however, sometimes the second three can come in handy so we mention them for the sake of completeness.

There is one more solution – called WinTunnel – that can be used, but as WinTunnel is not widely available outside Symbian we will not discuss it here.

13.3.1 Setting up the Easy Way – Using the SDK Configuration Dialogs

First, some good news. The SDK teams have created tools that will do a lot of the work for you! In the case of the S60 3rd edition emulator you

Table 13.2 Network connections for IP

Technology	Hardware required
WinPCap	A wired Ethernet card on your PC, ¹ and a network connection from that Ethernet card
WinTAP	Any IP connection from your PC, and the ability to enable Internet Connection Sharing on your PC
WinSock	Any IP connection from your PC
NT RAS	Two serial ports on your PC (or a second PC), the ability to enable NT RAS on the PC, and an IP connection from the PC
Analog modem	An analog modem
Mobile phone	A serial link (RS-232, Bluetooth or infrared) to a supported phone

already have the WinSock method set up for you. In the case of the UIQ 3.0 emulator, if you allow it to install WinPCap during installation, you'll have the WinPCap method available – although you still need to run the SDKConfig program provided, go to the Communications tab, and press 'Apply Ethernet' to set it up. In either case, this can save you a lot of work.

However, there will be circumstances where something isn't working due to your specific network or PC configuration. To deal with these cases, we'll offer some debugging tips below, in addition to showing you what the SDK tools are actually doing behind the scenes.

13.3.2 Configuring CommsDat

First we should mention a topic that affects you no matter which manual configuration method you use – configuring CommsDat.

CommsDat is the repository for all IP-related communications settings on Symbian OS and it also contains important information about default TSYs, and, for certain TSYs, configuration information.

Be warned – altering the data in CommsDat gives you a lot more freedom than using the UI configuration mechanisms, but it's also

¹ Note that most WLAN connections will fail to work with this method, as Ethernet packets are transmitted from Symbian OS in Ethernet II or 802.3 format which is incompatible with most WLAN chipsets. However, if the WLAN chipset in your machine happens to be of the (rare) type that expects Ethernet II or 802.3 frames from the operating system and performs the conversion to the 802.11 packet format internally, then this method will work. The best way to see if this works is to try it!

more error prone. You'll probably want to backup your `cccccc00.cre` file first just in case.

The CommsDat repository is stored as a binary central repository file in `\epoc32\wincsw\c\private\10202be9\persists\` and is called `cccccc00.cre`. The key tools to dealing with CommsDat are two Symbian OS executables that run under the emulator – the Comms database E`D`itor (CED) and the Comms database E`D`itor DUMPer (CEDDUMP).

CED is used to convert a text-based, human readable CFG file, containing CommsDat information, into a binary CommsDat repository. It can take a `-i` flag which allows you to specify the input file – note that this is the effective Symbian OS path rather than the Windows one, so to read a modified `cedout.cfg` file back in you should pass a path of `c:\cedout.cfg`. If no input option is specified it tries to read from `\epoc32\wincsw\c\ced.cfg`. Running CED can take some time, as the emulator has to start up first – on this machine it can take up to a minute.² At the end of the process CED leaves a log in `\epoc32\wincsw\c\ced.log`.

It is very important to check this log for errors. CED does not abort on failure – it tries to insert all information in the CFG file into the repository. Therefore the last line of the file might say 'SUCCESS' but individual records may not have been inserted. Make sure you check for failures to insert particular records – the error itself can be hidden amongst lots of other output, so errors such as 'Field count mismatch!' are easy to miss. The best way is often to check the number of records inserted into each table, and if it's unexpectedly low then check the preceding lines carefully for errors.

CEDDUMP performs the opposite process – taking a binary `cccccc00.cre` file and producing a `cedout.cfg` file in `\epoc32\wincsw\c\`. Dumping the original `cccccc00.cre` file is often a good point to start modifying settings in CommsDat.

² An unofficial, unsupported way of speeding this process up would be to add the string 'textshell', on a line of its own, to `\epoc32\data\epoc.ini`. That reduces the time taken to a few seconds. However, please be aware that this suggestion is both unofficial and unsupported, so it might change in the future – dependencies between CED and the components it uses may change, changes to startup mechanisms might require the emulator to boot fully, etc. So try it, but be warned – it might stop working at any time!

Don't forget to remove the 'textshell' string afterwards – the S60 emulator has problems drawing the console, and obviously neither UIQ nor S60 emulators boot to the user interface.

As we mentioned in Chapter 6, CommsDat is new for Symbian OS v9.1. Previously, information was stored in CommDB. The same tools are used to manipulate the data, but the binary file is different – in the case of CommDB it's called `cdbv3.dat` and lives in `\epoc32\winscw\c\system\data\`.

13.3.3 Setting up the Manual Way – Editing the Config Files

Using WinPCap and a wired Ethernet card

Using WinPCap and a virtual Ethernet adaptor allows you to share one physical Ethernet interface between the emulator and your PC. It does this by injecting packets through WinPCap into the bottom of the Windows networking stack, specifically at the link-layer ie. underneath the IP stack, into the top of the Ethernet driver. On the Symbian OS side, it appears as an Ethernet network interface.

Make sure you've read the previous footnote about using WinPCap – specifically that it is very unlikely to work with WLAN adaptors. You may be lucky and have an adaptor with which it is compatible, but the default recommendation is to use wired Ethernet adaptors only.

The Symbian OS virtual network adaptor doesn't show up as a separate network interface in Windows due to the way it is implemented – however, it can be treated in the same way as a virtual Ethernet interface, as it has its own properties – MAC address, and IP address – independent of the physical Ethernet interface that Windows is using.

This is the default IP connectivity option for the UIQ3 SDK. As a result, a compatible version of WinPCap is installed during the SDK installation. If you wish to use this method with S60 SDKs then you will need to download and install WinPCap yourself from www.winpcap.org. Obviously if you have the UIQ3 SDK on your machine then WinPCap is already installed and there is no need to do it again. At the time of writing, the compatible version of WinPCap is 3.0, although I'm using 3.1 beta 4 which is working successfully.

The configuration options for this virtual Ethernet adaptor are in the `epoc.ini` file, which is in `\epoc32\data`. The three lines of interest are:

- `ETHER_NIF` – this specifies which network card to use on your PC, and should be of the form `'\Device\NPF_{<long string of characters>}'`. The 'netcards' command from the SDK should list the possible interfaces and ask you to choose one – however, we often find it doesn't work properly. The other method is to use the 'Capture options' dialog in Ethereal, which has a drop-down box listing all the interfaces on your system. Pick the appropriate one, then copy the

correct part of the name (the part starting with '`\Device`'... into your `epoc.ini`.

- `ETHER_MAC` – this is the MAC address of your virtual Ethernet adaptor. Most commonly it is the MAC address of the interface specified in `ETHER_NIF`, but with the top byte changed from 00 to 02. This means that it's a 'locally administered' MAC address according to the IEEE, which also means that you're responsible for making sure it's unique! As long as you control the network card from which it is generated then this is a reasonable assumption, but be careful not to share this MAC address around – if there's more than one machine on the network with the same MAC address then you'll see very odd behavior from the IP networking components.
- `ETHER_SPEED` – this parameter doesn't actually do anything, so is normally left set to '10 Mbps'.

There is a script included with all SDKs called '`configchange.pl`'. Running this will attempt to set the configuration options above, then build a new CommsDat repository suitable for use with the virtual Ethernet adaptor.

One key point to note is that many business networks use some form of MAC address filtering. This means that when you generate your MAC address for the Symbian OS virtual network adaptor, you need to make sure it is permitted on your network. Talk to your local network administrator to find out whether you need to register the MAC addresses you use for the Symbian OS emulator with them. A typical symptom of this problem is that the emulator never gets an IP address over DHCP, or that DNS lookups fail with -5120 errors.

A great debugging tool for connection problems is Ethereal, available at www.ethereal.com. This also uses WinPCap, and allows you to see both PC and emulator traffic. You can also use a filter to display just the emulator traffic – just enter `eth.addr==<XX-XX-XX-XX-XX-XX>`, where `<XX-XX-XX-XX-XX-XX>` is the MAC address of your emulator, in the 'Filter' field.

Another potential issue you may encounter is the MTU of the virtual Ethernet adaptor. It is hardcoded to 1500 bytes, the most common value for Ethernet networks. However, some networks, especially some broadband networks in the UK, require a lower MTU. You'll know if this is the case if you have already had to alter the MTU on your PC to make it work with your network connection. Typical symptoms of this problem when using a web browser are that small pages download correctly, such as www.google.com, but the browser 'hangs' when downloading larger pages. Unfortunately, there is no way of altering the MTU without the Ethernet NIF source code, so currently there is no workaround for this problem. Keep an eye on the Symbian Developer website for updates on this issue.

Using WinTAP

Instructions for installing and using WinTAP are given as part of the download from Symbian's developer website at <http://developer.symbian.com/main/tools/devtools>.

The main point to make here is that WinTAP can solve many of the problems encountered when using the WinPCap method of connecting the emulator to a network, as it allows the use of Windows Internet Connection Sharing to connect the virtual network interface installed by WinTAP (this one is visible to Windows) to the PC's main network interface. However, that's also the drawback – the emulator does not appear on the main network, but instead has access through Internet Connection Sharing. This may cause some problems, depending on what level of network access your application is expecting.

Using WinSock

The WinSock connection method replaces the Symbian OS IP stack with a version that uses Windows native sockets directly (hence referring to it as the 'WinSock' connection method). Rather than use the Symbian OS TCP/IP stack and supporting infrastructure, WinSock maps sockets in Symbian OS directly to sockets on your PC.

The advantage of this is that your PC network connection is used directly – Symbian OS applications just appear to the Windows networking stack to be additional clients using TCP or UDP sockets, so there's no additional setup required. The disadvantages are that you can't have the same port listening in the emulator as on the PC (they are using the same TCP/IP stack), and that you are bypassing most of the Symbian OS networking stack, so the on-device behavior you see will be different. In particular, many of the features of the Symbian OS IP stack are not replicated in Winsock, so you will find that any protocol-level `GetOpt()`, `SetOpt()`, and `Ioctl()` calls you make will not work. Additionally, features such as multicast IP addresses are not supported by WinSock.

There are two versions of WinSock in existence. The original was created by Symbian, distributed from the Symbian Developer website, and targeted at UIQ2.1 SDKs – we won't talk about it here as it doesn't work with more recent Symbian OS releases. The other is a version from Nokia contained in the S60 3rd edition SDKs. It considerably enhances the version that Symbian created, and makes the connection appear more like a standard network connection on Symbian OS – for example, it uses the IAP selection dialog, and sends progress notifications when starting, just like a real network connection would.

If you just require a simple network connection for your testing then WinSock is ideal. It works 'out of the box' and provides a simple way to connect the emulator to an IP network with a minimum fuss. However,

be aware of the potential differences between the behavior of networking APIs when using WinSock and a real device.

Of course, if you're using the UIQ3 SDK then WinSock isn't available to you, so you'll have to consider another method.

Using NT RAS

NT RAS was once the mainstay of IP connectivity for Symbian OS development. More recently, Ethernet-based options have taken over as the connectivity methods of choice – they're faster, easier to set up and don't require extra cables. However, we'll briefly note down the settings for NT RAS here in case anyone's feeling particularly brave and wants to try using it.

The first requirement is a null modem cable between a serial port that Symbian OS can access and a COM port that is set up for NT RAS. The Windows side is fairly easy to set up by enabling the 'Routing and remote access' service, then configuring it through the 'Network connections' – 'Incoming connections' options. The Symbian OS side is harder, as it involves editing the entries in CommsDat to create a new IAP. See the section 'Configuring Commsdat' for more details on how to add and update entries in CommsDat. The UIQ3 SDK has an entry for NT RAS already (although it is set up to use a virtual IR serial port).

The key settings when using NT RAS are as follows:

In the `ModemBearer` record:

- `Agent` should be set to `'csd.agt'`
- `PortName` should be set to `'COMM:0'` (unless you want to use a virtual serial port, in which case you'd use `'BTCOMM:0'` or `'IRCOMM:0'`)
- `IfName` should be `'PPP'`
- `CSYName` should be `'ECUART'` (again, unless you're using a virtual serial port, in which case you'd want `'BTCOMM'` or `'IRCOMM'`)

In the `DialOutISP` record:

- `IfAuthName` and `IfAuthPass` should be set to appropriate values for a user on the machine that is hosting the RAS service. Traditionally Symbian has used `'RasUser'` and `'pass'` for a generic RAS account.
- `UseLoginScript` should be `'TRUE'`
- `LoginScript` should be `'CHARMAP \[windows-1252\]\nLOOP 10\n{\nSEND "CLIENTCLIENT"+<0x0d>\nWAIT 3\n{\n "SERVER"OK\n}\n}\nEXIT KErrNoAnswer$\n\nOK:\nEXIT\n'`

Note that the standard settings supplied by Symbian only have one 'CLIENT', but in order to get it to work with Windows 2000 and XP, the addition of a second 'CLIENT' is necessary (so the string sent becomes 'CLIENTCLIENT').

Using an analog modem

Using an analogue modem is very similar to using NT RAS, but this time the rest of the settings in the `ModemBearer` record are important, as is the phone number in the `DialOutISP` record. However, since we can't imagine there are many people out there who actually want to use this method for connecting to a network, we're going to leave the setup as an exercise for the reader.

Using a serial link to a mobile phone

This is covered in the next section as part of the wider setup for attaching a mobile phone to the emulator.

13.4 Telephony

Attaching a phone to the Symbian OS emulator is possible, as Symbian provides a TSY that uses AT commands to communicate with a phone over a serial cable or a virtual serial port of some kind. The `MultiModeTSY` (`MM.TSY`) is available in both UIQ and S60 SDKs.

Finding a phone that supports the required range of AT commands though is a somewhat trickier prospect. Fewer and fewer recent phones support a sufficient set of AT commands to allow the `MM.TSY` to initialise, let alone operate. The best strategy is to try different phones until you find one that works. The rule of thumb is the older the better, and the more basic the better.

In order to use a phone with the `MM.TSY`, you firstly need to connect to it. This should be possible over infrared, Bluetooth or a serial cable (if your phone supports it). Once again, configuration involves changing values in `CommsDat`.

Firstly, dump your current `CommsDat` repository to CED format using the instructions in the 'Configuring `Commsdat`' section of this chapter. Then you need to alter the following sections:

In the `ModemBearer` table, find a suitable modem to use. In most cases you can just pick the first one. Alter the `TSYName` to 'MM' and the `PortName` and `CSYName` to whichever technology you want to use to connect – Bluetooth, infrared or serial (see 'Using NT RAS' for more details on the options for these fields). Finally, in order to use the phone for GPRS access, change the `Agent` field to 'psd.agt'. To use it for circuit switched access, change the `Agent` field to 'csd.agt'.

In the `GlobalSettings` table, change `ModemForPhoneServicesAndSms` and the `TsyForBearerAvailability` to 'MM'.

Due to our inability to find a recent phone that supported MM TSY, we don't have steps for the rest of the changes that would be necessary to use a phone with the emulator. At this point, the setup should be close to working – SMSs and MMSs should be routed to the TSY, and incoming messages should be sent to the message store. The IP networking code should also be able to use GPRS to connect to the network. However, as we haven't got this far ourselves, there may be some areas which we've failed to cover where the configuration needs tweaking some more.

13.5 'Help, help, my serial port's been stolen'

One of the most common problems in setting up the emulator is a serial port being in use by another component within Symbian OS. The best solution to this is to download the excellent Portmon from Sysinternals (now Microsoft) at <http://www.microsoft.com/technet/sysinternals/ProcessesAndThreads/Portmon.msp>. Then have a look at the traffic that's being sent over the serial port.

Typically, you can spot the TSY using the port, as the traffic will have AT commands in the output. Switching the viewing mode from ASCII to hex is the best option to spot BCSP or IrDA traffic.

Bluetooth BCSP traffic is easy to spot with a repeated pattern of C0 40 41 00 7E DA DC ED ED A9 7A C0 being transmitted. IrDA traffic should show up as 10 FF or C0 bytes in a row.

If there is no IrDA pod attached, the other way to tell that IrDA is trying to use the port is to look for a sequence of about 12 attempts in total to set or clear RTS or DTR – you can identify these as they look similar to 'IOCTL_SERIAL_SET_DTR' (but might say RTS instead of DTR or CLR instead of SET). This is the IrDA stack in Symbian OS trying to communicate with the pod, and results in a very characteristic sequence of events in the Portmon window.

```
0.00005671 epoc.exe      IRP_MJ_CREATE  Serial0SUCCESS Options: Open
0.00000782 epoc.exe      IOCTL_SERIAL_GET_COMMSTATUS
Serial0SUCCESS
0.00000140 epoc.exe      IOCTL_SERIAL_GET_BAUD_RATE
Serial0SUCCESS
0.00000168 epoc.exe      IOCTL_SERIAL_GET_LINE_CONTROL
Serial0SUCCESS
0.00000140 epoc.exe      IOCTL_SERIAL_GET_CHARSSerial0 SUCCESS
0.00000112 epoc.exe      IOCTL_SERIAL_GET_HANDFLOW
Serial0SUCCESS
0.00000503 epoc.exe      IOCTL_SERIAL_GET_COMMSTATUS
Serial0SUCCESS
```

```

0.00000419 epoc.exe IOCTL_SERIAL_GET_COMMSTATUS
Serial0SUCCESS
0.00000531 epoc.exe IOCTL_SERIAL_SET_WAIT_MASK
Serial0SUCCESS Mask: RXCHAR CTS DSR RLSD BRK ERR RING
0.00000140 epoc.exe IOCTL_SERIAL_SET_QUEUE_SIZE
Serial0SUCCESS InSize: 1024 OutSize: 1024
0.00000447 epoc.exe IOCTL_SERIAL_GET_COMMSTATUS
Serial0SUCCESS
0.00000531 epoc.exe IOCTL_SERIAL_GET_COMMSTATUS
Serial0SUCCESS
0.00000140 epoc.exe IOCTL_SERIAL_GET_BAUD_RATE
Serial0SUCCESS
0.00000140 epoc.exe IOCTL_SERIAL_GET_LINE_CONTROL
Serial0SUCCESS
0.00000140 epoc.exe IOCTL_SERIAL_GET_CHARSSerial0 SUCCESS
0.00000140 epoc.exe IOCTL_SERIAL_GET_HANDFLOW
Serial0SUCCESS
0.00000419 epoc.exe IOCTL_SERIAL_GET_COMMSTATUS
Serial0SUCCESS
0.00000112 epoc.exe IOCTL_SERIAL_GET_BAUD_RATE
Serial0SUCCESS
0.00000140 epoc.exe IOCTL_SERIAL_GET_LINE_CONTROL
Serial0SUCCESS
0.00000112 epoc.exe IOCTL_SERIAL_GET_CHARSSerial0 SUCCESS
0.00000112 epoc.exe IOCTL_SERIAL_GET_HANDFLOW
Serial0SUCCESS
0.00001090 epoc.exe IOCTL_SERIAL_SET_BAUD_RATE
Serial0SUCCESS Rate: 960
0.00000670 epoc.exe IOCTL_SERIAL_SET_DTR Serial0 SUCCESS
0.00000643 epoc.exe IOCTL_SERIAL_SET_LINE_CONTROL
Serial0SUCCESS StopBits: 1 Parity: NONE WordLength: 8
0.00000447 epoc.exe IOCTL_SERIAL_SET_CHAR Serial0 SUCCESEOF:0
ERR:0 BRK:0 EVT:0 XON:11 XOFF:13
0.00000615 epoc.exe IOCTL_SERIAL_SET_HANDFLOW
Serial0SUCCESS Shake:80000009 Replace:80 XonLimit:768 XoffLimit:256
0.00000475 epoc.exe IOCTL_SERIAL_GET_COMMSTATUS
Serial0SUCCESS
0.00000447 epoc.exe IOCTL_SERIAL_GET_COMMSTATUS
Serial0SUCCESS
0.00000615 epoc.exe IOCTL_SERIAL_GET_MODEMSTATUS
Serial0SUCCESS
0.00000643 epoc.exe IOCTL_SERIAL_GET_COMMSTATUS
Serial0SUCCESS
5.00801834 epoc.exe IOCTL_SERIAL_WAIT_ON_MASK
Serial0SUCCESS
0.00000447 epoc.exe IOCTL_SERIAL_GET_COMMSTATUS
Serial0SUCCESS
0.00000587 epoc.exe IOCTL_SERIAL_GET_MODEMSTATUS
Serial0SUCCESS
0.00000978 epoc.exe IOCTL_SERIAL_GET_COMMSTATUS
Serial0SUCCESS
0.00002263 epoc.exe IOCTL_SERIAL_SET_WAIT_MASK
Serial0SUCCESS Mask: RXCHAR CTS DSR RLSD BRK ERR
0.00000251 epoc.exe IRP_MJ_CLEANUP Serial0SUCCESS
0.01102067 epoc.exe IRP_MJ_CLOSE Serial0SUCCESS

```

Example 13.1 A component within Symbian OS opening the COM port to check for an attached device

13.5.1 Freeing up Serial Ports that are in Use

In order to stop Bluetooth or IrDA using a serial port, the best option is to alter the setting in the appropriate. `esk` file (`bt.bt.esk` for Bluetooth, `ir.irda.esk` for IrDA) to set the `'port= '` field to an unused COMM port. Preventing the TSY from using the COMM port is often harder. Symbian has internal mechanisms for preventing the telephony subsystem from starting up, mainly by preventing `'watcher.exe'` from loading, but the additional dependencies introduced in the UI platform SDKs make this task more difficult. Altering the `PortName` fields in the `ModemBearer` records in `CommsDat` can help, but this depends on how the TSY that is in use gets the information about which port to use.

13.6 Summary

In this chapter we've:

- covered how to set up the Symbian OS emulator for a variety of communications development tasks
- seen both the standard methods that can be used in most cases to configure the emulator, as well as a lot of advanced information about manually tweaking configurations
- discussed a few of the common problems that can occur with each of the setups, and some possible fixes for them.

14

The Future

As you've probably noticed, the rate of change in the world of mobile communications is rapid. Making predictions about the future can often end up making you look foolish later when something totally different happens; and making *accurate* predictions is hard, as many technology pundits have found out over the years. So we're not going to claim that our predictions are accurate or sensible! However, we're still going to rush headlong into picking some interesting future technologies that might turn up in Symbian OS, or problems that need solving in today's communications environments. We should say at this point that this is in no way an official list, and Symbian may not be working on technologies in these areas – this is strictly a list drawn up by the authors of this book based on our own experiences. (Just to make it clear who's responsible when it turns out we were completely wrong!)

We've divided the list up into three main areas – networks, services and interaction. By networks, we mean the underlying networks that actually move the data around. By services, we mean the applications you can use over a network to accomplish something that you want to do. By interaction, we mean better ways to actually make use of the networks to get to the services. In each case we'll pick an example or two that illustrates the work being done in these areas, or the problems that remain to be solved.

14.1 Better Networks

One of the most obvious trends, which shows no signs of stopping, is the quest for more bandwidth. Each time a new technology is released – wired or wireless – the next generation of services manage to step up their bandwidth requirements to match. (Perhaps the other way to look at this is that the previous generation of networks didn't offer enough bandwidth to the services that used them!)

One topic closely related to bandwidth is the more general issue of quality of service (QoS). Of course, if you've got plenty of bandwidth then you've no need for any QoS mechanisms – if there's never a decision to be made between which of two (or more) packets to send next you have neither the need for, nor the ability to apply, QoS. (Of course, you also have the perfect network – so be prepared to make a lot of money from selling the technology!) This is QoS through overprovisioning, and depending on which technology you're deploying can either be extremely cost effective, or prohibitively expensive.

In wireless technologies bandwidth is often at a premium, and we therefore need to take a more sophisticated approach. Offering reliable services means deploying QoS solutions in our wireless technologies. These exist today for many technologies, so really what we're expecting to see is their move to the mainstream as they become more and more necessary to allow networks to differentiate between the ever-increasing amount of traffic flowing over them. And we're not just talking about cellular networks here, but wireless LAN and Bluetooth as well, along with any future wireless technologies that come to market.

Let's have a quick look at one example which is currently offering the potential for a significant increase in available bandwidth – the addition of a UWB bearer to the Bluetooth specifications.

14.1.1 Bluetooth and UWB

The Bluetooth SIG have announced their intention use ultra-wideband (UWB) radios in a future version of the Bluetooth specification, alongside the traditional 2.4 GHz radios in use today. This offers a potentially massive increase in bandwidth available to Bluetooth applications – some current figures suggest at least 110 Mb/s, up from a mere 723 Kb/s in the original Bluetooth specifications and up to 2.1 Mb/s today. This opens up several new use cases for Bluetooth devices that were previously problematic, or impossible to achieve. New applications could include high-quality video streaming between your phone and your TV, much reduced times for bulk data transfer to synchronize your music with your handheld device, and allowing Bluetooth PAN profile to effectively compete with WLAN in data rate terms.

The question mark remains over battery life though – is it possible for Bluetooth to maintain its reputation as a low power technology whilst at the same time moving to a higher data rate? To some extent, such questions are irrelevant. The old radio standard will remain in devices alongside the new one, allowing the selection of the appropriate radio based on the application's requirements. In some ways this is the ideal situation, even though it complicates the engineering of the Bluetooth chipset, as it means that Bluetooth can address a wider range of applications, from low data-rate systems that need to minimize power

consumption (such as HID devices – keyboards, mice and the like) to high bandwidth applications such as high resolution video streaming. It also gives Bluetooth a clear advantage over wireless USB, which currently only has the single, higher data-rate physical layer – without the fallback option that Bluetooth offers.

14.2 Better Interaction

With ever-increasing numbers of wireless technologies, and an even greater number of use cases, the problem of simple interaction with the different technologies is becoming harder and harder. Interacting with several of them at the same time is practically a nightmare! This is an area where much work is going to have to be done to make the use of the technology as transparent as possible.

There are two areas we want to focus on – interacting with a single technology in various use cases, and interacting with several technologies to achieve a single use case. Let's look at the single technology, multiple use case scenario first, and compare and contrast some current technologies – Bluetooth and TCP/IP.

Bluetooth excels in the environment in which it operates – short-range communication. It has excellent support for discovering devices and services in the vicinity. That support has three essential parts: the ability to present a list of devices in the vicinity, which is essential on devices with limited input capabilities as typing in names is much harder than picking from a list; the ability to query those devices to discover what services they offer; and the fact that the radio has limited range allows a clear understanding of the distance of the discovered device from your current location. And perhaps just as importantly, these services are standardized for all Bluetooth devices.

IP also excels in its natural environment. It can connect machines on opposite sides of the planet, scales to hundreds of millions of machines being connected to the network, and can be made fault tolerant to allow parts of a network to fail and the whole system to keep running. Naturally, it does not offer the native ability to produce a list of all the devices attached to the network – that too would be something of a usability nightmare!

Now we have the problem of using either system outside the environment in which it was conceived – or, perhaps worse, combining the two systems together into one – which is exactly what Bluetooth PAN profile sets out to achieve. There is clearly some work to be done here in integrating the two technologies so that they fit together well (in fact the same point is true for using WLAN in an ad-hoc fashion on phones and other non-PC devices). No one wants to type awkward names, such as 'My Nokia N95', or 'SonyEricsson P990' into their phone to be able to

connect to a service offered by another local device – they don't have to do this with traditional Bluetooth profiles, but anything offered over PAN or WLAN can't use a standard service discovery mechanism. (In fact, they can use any number of standardized service discovery mechanisms, just not a single standard one). There's clearly room for improvement here.

Now let's consider the one use case, multiple technologies scenario, but since there's more of an answer there rather than a series of problems, we'll pull it out into its own subsection.

14.2.1 Automatic Bearer Selection and Bearer Mobility

As we've seen in this book, when you want an IP connection, the number of possible technologies that can provide this on a mobile device is significant, and increasing. Whilst there will always be applications that only wish to connect using a very specific technology, indeed a specific access point – MMS being a prime example – most applications would benefit from the ability to choose between a set of access points when connecting to a service; and it would be even better if they received notifications about new access points becoming available so they could consider switching. If the application wasn't even aware of the change of bearer, it would be ideal!

The utopia of seamless roaming offered by technologies such as mobile IP seems to have been slow to arrive on any major scale, so is there another step we can take on the way to seamless roaming? Or, indeed, is a non-seamless roaming system good enough for end-user needs?

Well the first step must be to try deploying a system that offers this ability to applications – when starting a connection, make a dynamic selection between access points based on their availability; and provide some form of indication of availability of alternate access points during the application's lifetime. In fact, both the E series phones from Nokia and the UIQ3 phones from Sony Ericsson contain the first of these technologies – the ability to try groups of access points when connecting to allow a more dynamic selection routine.

14.3 Better Services

Services are the hardest part of this mix to try and predict – so we're not even going to try. By focusing on the other two parts – networks and interaction – Symbian OS enables you to create the next great service. A solid foundation of easy-to-use technology is necessary to explore the next step in services – and that's what Symbian OS aims to provide.

14.4 The End

That's it for the book. We hope you've found it useful. Symbian Press runs surveys from time to time to gather feedback on the various books that are published, so if you'd like to leave us some feedback, keep an eye on <http://developer.symbian.com/main/academy/press> for the next survey.

Appendix A:

Web Resources

Bluetooth

www.bluetooth.org

Forum Nokia

www.forum.nokia.com

Infrared Data Association

www.irda.org

Internet Engineering Task Force

www.ietf.org

Open Mobile Alliance

www.openmobilealliance.org

Symbian Developer Network

<http://developer.symbian.com>

Symbian OS Documentation

<http://developer.symbian.com/main/oslibrary>

Symbian Press

<http://developer.symbian.com/main/academy/press>

Telephony: 3GPP

<http://www.3gpp.org>

UIQ Developer Community Portal

<http://developer.uiq.com>

Appendix B:

Authorizing FlickrMTM to Use Your Flickr Account

The current way of authorizing any mobile application to use Flickr is to use your desktop browser to log into a Flickr URL which authorizes the application. The web-based authorization produces a temporary code which needs to be input into the mobile application as a one-off. The mobile application will now have access to upload to your account.

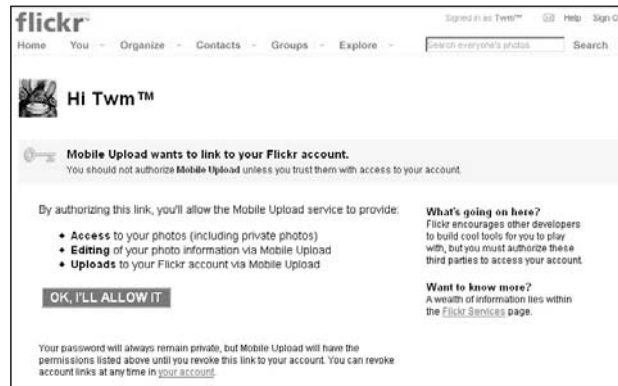
Without this authorization FlickrMTM will always fail with 'Sending Failed'.

Step 1 – Get a Flickr Account

In order to upload images to Flickr, you must have a Flickr account which can be obtained from www.flickr.com. The basic account is free of charge and can be used with the FlickrMTM.

Step 2 – Authorize FlickrMTM over the Web

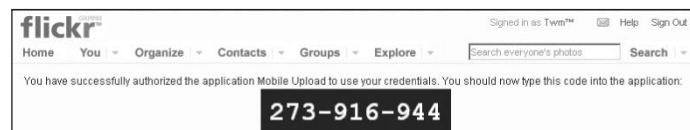
With your desktop browser (or suitably powerful mobile browser), visit the following URL (signing in if necessary) in order to authorize FlickrU-upload.dll to upload photos to *your* account: www.flickr.com/auth-35520. You will be presented with a page such as this.



Click on "OK, I'll allow it"



And you will get a personal authorization code.

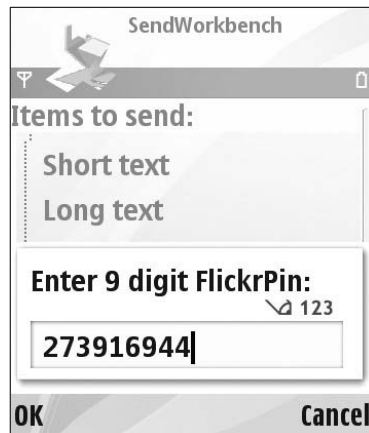
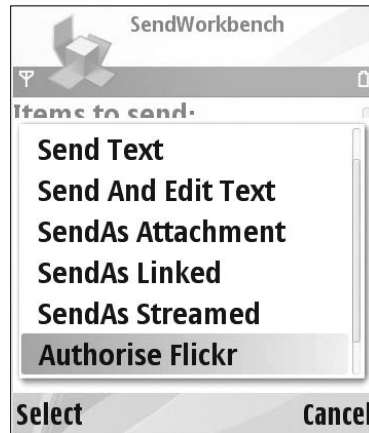


Your number will be different.

Step 3 – Enter your Authorization Code on the Device

You will need to authorize each emulator or device which you wish to use for uploading to Flickr. You must repeat Step 2 for each device since the authorization code expires once authorization has been completed.

Open the SendWorkBench application and select the 'Authorise Flickr' option.



Appendix C:

SendWorkBench.app Guide

The SendWorkBench application serves as a test application for driving the FlickrMTM and as an example for demonstrating the use of the built-in SMS MTM message type.

The main display shows four different types of data. This is the data type which will be used when selecting 'Send' from the menu. For example, by moving the cursor to 'Long text', the send dialog will only list MTMs which support body text with a length of more than 255 characters.



Menu item	Information
Send	Brings up the send menu to offer the user a choice of send bearers which can send the item currently highlighted in the main application window
Send Text	Sends a text using RSendAs
Send And Edit Text	Brings up the SMS editor with a populated message using RSendAs.
SendAs Attachment	Use RSendAs to send sendasexample.jpeg using the FlickrMTM
SendAs Linked	Use RSendAs to send sendasexample.jpeg using a link to the file using FlickrMTM
SendAs Streamed	Use RSendAs to send sendasexample.jpeg by writing the data into the mail store, and then sending it using FlickrMTM
Authorise Flickr	Authorize FlickrMTM to use upload to your Flickr account (see Appendix B)
Exit	Close SendWorkbench

Index

- 2.5G networks 13, 156, 358
 - see also* GERAN
- 3G networks 13, 156, 197–203, 211, 358
 - see also* UTRAN
- 3GPP specification 157, 197, 201, 205, 413
- A2DP, Bluetooth 12, 65, 81, 120

- Abort 314
- Abort command, OBEX 274–5, 279–80, 286–7, 289, 314, 332, 339–41
- AbortIndication 332, 339–41
- abstract concepts, messaging 255–6
- Accept 36–7, 58, 146–8
- access point names (APNs), GPRS 156–9, 162–3, 167–9
- ACL *see* asynchronous transport
- Actisys pods 395–6
- ActivateSniff-Requester 92–3
- active connections
 - information 196
 - sockets 32, 33–6, 56–62, 196
- active mode, physical links 71–2, 92–3
- active objects 6–7, 55–62, 102–3, 131–2, 208–9, 233–4, 268, 305, 311–12, 316–17
 - see also* engines; events
 - messaging 233–4
 - problems 59–60
- active scheduler 60
- adapters, Device Management system (DM) 367–87
- Add 141–3, 203
- Add command, OMA Device Management system 384–5
- AddAttachment 249–50, 255
- AddExtensionSetL 201–3
- AddFilter 363
- AddHeaderL 293–8, 311–16
- AddHttpL 292–3
- AdditionalProtocol-DescriptorList 82
- AddLinkedAttachment 250
- AddObserverL 234–5
- addresses
 - ESOCK 50–2, 54, 86
 - Internet 31, 54–6, 176–7
 - IP addresses 31, 50, 54–6, 147–8, 159, 176–80, 185–6
 - sockets 31, 33–4, 50–2, 54–5, 86, 97, 127–31, 133–40
- ADSL connections 358–9
- advertising displays 69
- AES-based cipher suites 12
- agents, SDDB 106–9, 112–18
- AGT 18
- AIF files 262
- aInboxEntry 228–9
- AllowRoleSwitch 92–3
- aMessageType 228–9
- aMsvId 227–9
- aMsvSession 227–9
- analog modems 397, 403
- aOrdering 227–9
- APIs 3–9, 19–21, 27–62, 130, 155, 159–60, 162–204, 205–6, 218, 220, 226–32, 241–71, 290–342
 - partner-only APIs 5–6
 - RConnection 8–9, 46–50, 159–60, 162–204, 205–6, 345–9, 360–1
- APNs (access point names), GPRS 156–9, 162–3, 167–9
- AppendAttachmentL 252–3
- appendices 413–20
- applications 3–12, 407–11
 - see also* code
 - example applications 9
 - future prospects 407, 410–11
 - QoS 156, 158, 197–203, 408
 - testing 7

- ARM 10
- AsByteSeq 295–6
- ASCII 135–6, 141–3, 404
- ASSERT 116–18, 261, 324
- AssertIdle 265–6
- association concepts, Bluetooth 66–7
- asterisk wildcard 131–2
- asynchronous calls
 - cancellations 185–6
 - messaging 233, 236–7, 248, 267
 - sockets 34–9, 42–5, 55–6, 131–2, 151, 182, 185–6, 195
- asynchronous server API, OBEX 338–41
- asynchronous transport (ACL)
 - see also* L2CAP
 - concepts 65, 66, 72–4, 78–82
- Attach 48–9, 173
- AttachmentManagerL 226
- attachments
 - files 248–50, 252–3, 254–5, 263–5
 - messaging concepts 223–6, 245–6, 248–50, 252–3, 254–5, 263–5
- AttributeRequest... 106–7, 116–18
- attributes, SDP 74–8, 80, 104–9
- authentication 12, 22, 67–8, 71–2, 97–103, 186–93, 275–8, 282–4, 289, 310–12, 333–42, 344, 359–62
 - Bluetooth 67–8, 71–2
 - HTTP 344, 359–62
 - OBEX 275–8, 282–4, 289, 310–12, 333–42
- authorization concepts 72, 97–103, 186–93
- AV (audio video) protocols 17, 33, 65, 83, 118–24
- AvailableCipherSuites 186–93
- AVCTP (audio video control transport protocol) 17, 33, 83, 118–24
- AVDTP (audio video distribution transport protocol) 17, 33, 65, 83
- AVRCP (audio video remote control protocol) 83, 118–24
- BAFL APIs 353
 - see also* Stringpool
- bandwidth 61–2, 71, 197–203, 407–9
- baseband *see* signalling stack
- batteries 10–11, 41, 63–6, 70–1, 91–3, 408–9
- BC (binary compatibility) 13–14
- BCSP protocol 391–4, 404
- bearer technologies 160–2, 163–75, 193–6, 244–5, 391–2, 410
 - different bearers 172–3, 193–6, 244–5, 391–2, 410
 - disconnections 175
 - IP networks 160–2, 163–75, 193–6, 392, 410
 - progress notifications 193–6
- billing practices, GPRS 157
- binary compatibility (BC) 13–14
- Bind 36–7, 183–6
- blocked operations 38, 40–2
- Bluetooth 4–20, 27–8, 33, 50, 53, 63–124, 126, 160–1, 164–5, 243, 245–6, 265, 273–4, 278, 286, 287–9, 305–9, 318–22, 330, 336, 391–4, 403–6, 408–10, 413
 - A2DP 12, 65, 81, 120
 - ACL 65, 66, 72–4
 - advantages 64, 408–10
 - architectural overview 65–6
 - association concepts 66–7
 - authentication 67–8, 71–2
 - AV protocols 65, 83, 118–24
 - batteries 63–6, 91–3, 408–9
 - cables 63–4
 - channel architecture 66
 - CoD (Class of Device) 68, 88–91, 112–18, 288, 318
 - concepts 11–20, 33, 50, 53, 63–124, 126, 160–1, 164–5, 243, 245–6, 265, 273–4, 278, 286, 287–9, 305–8, 309, 318–22, 330, 336, 392
 - connectability concept 66–8, 83–91, 126, 319–20
 - controller 17, 65–6, 82–3, 121–4
 - definition 63–4
 - device class 68–9, 88–91, 112–18, 288, 318–22
 - device discovery 66–9, 83–91, 112–18, 287–8, 305–8, 318–22, 409–10
 - device names 50
 - discoverability concept 66–9, 83–91, 112–18, 287–8, 305–8, 318–22, 336, 409–10
 - emulator 391–4, 403–6
 - error codes 111
 - ESOCK 83
 - example application 112–18, 123–4
 - FTP servers 273, 336–7
 - functionality levels 64
 - future prospects 408–10
 - high-level architecture 17
 - historical background 63–4
 - host 17, 65–6, 82–3
 - IP 155, 160–1, 164–5, 409–10
 - IrDA contrasts 126
 - L2CAP 64–6, 72–4, 78–84, 93–6, 102–3, 105–6, 111, 114–24
 - logical transports 66, 71–4, 91, 114–18
 - OBEX 20, 21–2, 65, 69, 76, 83, 89–91, 105–6, 245–6, 256, 273–4, 278, 286, 287–9, 305–8, 309, 318–22, 330, 336, 392
 - operating band 63–4
 - OPP 308, 336–7
 - pairability concept 67–8, 88–9, 278

- PAN profile 11–12, 16–17, 65, 83, 90, 155, 164–5, 167, 392–5, 408–10
- passkeys 68, 97–8
- physical links 66, 68–72, 91–3, 112–18
- piconet topology 66, 69–72
- ports 80–2, 391–4, 397, 404
- power management 70–2, 91–3, 408–9
- profiles 11–12, 16–17, 64–5, 81, 118–24
- protocols 11–12, 16–17, 64–6, 73–4, 118–24
- registry 109–11
- remote control 83, 118–24
- RFCOMM 74–6, 78–83, 96, 102–3, 105–6, 116–18, 274, 288, 308, 309, 318
- S60 84, 86–8, 98–100, 243, 392–4
- SCO/eSCO 65, 66, 72–3
- SDAP 109, 112–18
- SDDBs 75–8, 104–9, 318–19
- SDP 68, 74–8, 80–2, 83, 90, 103–9, 112–18, 278, 287–8, 307–8, 318–19, 336–7
- security issues 71–2, 97–103, 278, 319–20, 338
- service class 68–9, 74–8, 89–91, 104–9, 112–18
- service discovery 68–9, 74–8, 80–2, 83, 86, 90, 103–9, 112–18, 287–8, 305–8, 318–22, 336, 409–10
- sockets 83, 93–4, 96–103, 114–18
- speed statistics 126
- standards 64–6, 73–4
- Symbian OS 82–124, 391–4
- technology overview 63–82
- transport class 66, 91, 114–18
- UIQ 84, 86–8, 98–100, 393
- USB 391, 394, 409
- UWB 408–9
- v2 architecture 11–12
- visible devices 66–8
- BNEP 83
- body data, HTTP 348–53
- Body headers, OBEX 276–7, 281, 291–304, 309–16, 321–41
- books 6
- BSD sockets 180
- BTCOMM 402–3
- BTLinkManager 84
- bt_sock.h 111
- buffers 55, 59, 61–2, 289–304, 308–16, 322–6
 - concepts 55, 59, 61–2
 - OBEX 289, 291–304, 307–16, 322–6
- builder patterns 109
- built-in applications 166, 213, 220–40
- bus topology 112–18
- byte-oriented data channels
 - see also* stream sockets
 - concepts 30–1
- C++ 3–4, 162
- C (heap-allocated) classes 6, 102, 293–4
- c32 serial server 10, 15
- cables, Bluetooth 63–4
- CActive 54–8, 85–8, 102–3, 137–40, 145–8, 208–9, 220–1, 233, 268, 290–1
- Cambridge Silicon Radio (CSR) 393
- camera phones 11
- Cancel 265
- CancelConnect 32, 36
- CancelHandshake 186–93
- CancelIndication-Callback 339–41
- CancelIoctl 42
- cancellations
 - asynchronous calls 185–6
 - HTTP 348–9, 353
 - sockets 32, 36, 37–9, 42–5, 185–6
- CancelProgress-Notification 47–8
- CancelRecv 39–42, 186, 191–3
- CancelSend 37–9, 186, 191–3
- CanCreateEntryL 259–71
- capabilities 13, 29, 241–2, 247, 252–5, 260–2, 268–9, 361
- CAPABILITY 13
- CAppView 208–9
- case 54–5, 298–9
- Casira pods 393
- CBase 114–18, 228, 293
- CBaseMtm 259
- CBaseMtmUi 259
- CBaseMtmUiData 259
- CBaseServerMtm 259
- CBluetoothSocket 93–4, 96–7, 102–3, 113–18
- CBTDevice 110–11
- CBTDeviceDiscoverer 85–91
- CBTRegistryResponse 110–11
- CBufFlat 301–4
- CCDIARecord 168–70
- CCertificate 348–9
- CCnvCharacterSet-Converter 136–40
- CCommsDatabase 167
- CD 129
- cdbcols.h 196
- cdbv3.dat 399
- CDialer 208–9
- CDMA 205
- CED 398, 403–4
- CEDDUMP 398
- CEikonEnv 244, 252–3
 - see also* Send dialogs
- cell IDs 206–7
- cellular modem *see* signalling stack
- Central Repository 223
- certificates, secure sockets 186–93, 348–9
- CF (compact flash) 393
- CFG files 398
- CFlickrClientMtm 259
- CFlickrDataMtm 259
- CFlickrRestAPI 266–7
- CFlickrMtmClient 259
- CFlickrMtmServer 259
- CFlickrMtmUi 259, 262–3
- CFlickrMtmUiData 259
- CFlickrRestrAPI 344, 346, 349, 354–5
- CFlickrServerMtm 259, 265–6
- CFlickrUIMtm 259

- CFlickServerMtm 266–8
- CGetter 208–9
- CGetWebPage 53–62
- ChangeEntryL 234
- chapters, reading guide 9
- character sets 134–43
- ChildDataL 229–30
- ChildrenL 229
- ChildrenWithMtmL 229
- ChildURLListL 384
- CHttpConnectionInfo 346–7
- chunked transfer encoding 22
- CIdentifiableSocket 114–18
- CImageUploadOperation 263
- CImEmailMessage 225–6, 231
- CImHeader 230–2
- CInfiniBTEngine 118
- CInformer 208–9
- cipher suites 12, 186–93
- circuit-switched networks 28, 205
 - see also* CSD; GSM
 - concepts 28, 205
- CIrDeregisterLocal-Service 144
- CIrDeviceDiscoverer 130–40
- CIrInboundConnection 147–8
- CIrOutboundConnection 145–6
- CIrServiceDiscoverer 137–40
- Class of Device (CoD), Bluetooth 68, 88–91, 112–18, 288, 318
- classes
 - see also* C...; M...; R...; T...
 - FlickrMTM 258–9
 - HTTP 344–64
 - IP 162–3
 - OBEX 290–1, 293–306, 316–33
 - Send dialogs 250–5
 - types 6
 - wrapper classes 27, 51, 52, 55–62, 96–7, 102–3
- CleanupStack::Pop 297–8, 311–13
- CleanupStack::PopAnd-Destroy 115, 170, 247, 250, 255, 355
- CleanupStack::PushL 115, 168–70, 238, 247, 249, 295–9, 311–12
- ClearHintBitL 142–3
- CLIENT 402–3
- client APIs
 - HTTP 344–64
 - OBEX 290, 304–16, 333–42
- client MTMs 20–1, 220–3, 230–1, 246–50, 256–71
 - see also* message-type modules
 - concepts 246–50, 256–71
 - server MTMs 264–5
- client sockets, IrDA 144–6
- client-server framework 6, 21–2, 29–62, 144–8, 220–2, 246–50, 274–9, 290, 304–42, 343–64
 - see also* server...
- ClientCert 186–93
- Close 32, 36, 37, 42–5, 46–50, 174–5, 182, 191–3, 249–50, 345–7, 353
- closing, sockets 36, 37, 42–5, 62, 174–5, 182, 191–3, 194–6
- CMDBRecordBase 168–70
- CMDBRecordSet 168–70
- CMDBSession 168–9
- CMessageData 252–3
- CMessageSummaryEngine 218–33
- CMessageSummaryGenerator 218–21, 224–32
- CMsvEntry 226–35
- CMsvOperation 262–5, 267–8
- CMsvSession 221–2, 229–31, 234–5, 246–50, 256–71
- CMsvStore 225–6, 230–2
- CMsvStream 225
- CMyObexClientApp 311–16
- COBEX 330–2, 334, 337
- COBEXBaseObject 290–304, 310–16, 328–30
- COBEXBufObject 290–304, 322–30, 339–41
- COBEXClient 304–16, 333–41
- COBEXFileObject 290–304
- COBEXFtpServer 322–30
- COBEXHeader 291–304, 311–16
- COBEXHeaderSet 291–304, 315, 324–6
- COBEXNullObject 304
- COBEXServer 316–33, 334–42
- CObservers 208–9
- CoD (Class of Device), Bluetooth 68, 88–91, 112–18, 288, 318
- code
 - see also* applications
 - downloads 9
 - practical examples 52–62
 - sample code 8
- COMM ports 391–406
- commands
 - OBEX client applications 274–5, 279–80, 281, 282–7, 289, 305–6, 310–16
 - OMA Device Management system 384–7
- CommDB 162
- CommDbConnPref.h 170
- CommitAtomicL 385–6
- comms provider modules (CPMs), concepts 15
- Comms System plug-ins (CSYs), concepts 14, 17, 19, 402–4
- CommsDat 47, 161–3, 167–70, 174–80, 196, 205–6, 347, 355, 360–1, 397–9, 402–4, 406
- CommsDatTypesV1_.h 169
- communications 4–12, 13–23, 28, 407–11
 - see also* Bluetooth; connectivity; HTTP; IP; IrDA; messaging; Object EXchange; telephony
 - buffers 55, 59, 61–2
 - concepts 4–12, 13–23, 28, 407–11
 - future prospects 407–11
 - high-level overview 13–23
 - historical background 10–21
 - key frameworks 14–19
 - options 28, 49–50, 89–91, 155, 158, 160–1
 - overview 13–23, 28
 - performance measure 61–2, 71, 197–203

- protocol types 31, 32–3, 80–2
- signalling stack 13–14
- compact flash (CF) 393
- Complete 232
- CompleteOutstandingCmdsL 386
- configuration settings 22–3, 391–406
- ConfigureChannelPriority 94–5
- ConfigureReliableChannel 94–5
- ConfigureUnreliable-Channel 94–5
- Connect 29, 32, 33–8, 54–5, 58–62, 94–5, 145–6, 176–7, 183–6, 310–12, 315, 336–7, 346
- Connect command, OBEX 274–5, 279–80, 281, 282–7, 289, 310–12, 315, 321–2, 333, 335–6, 339–40
- connectability concept, Bluetooth 66–8, 83–91, 319–20
- connecting
 - Bluetooth devices 66–8, 83–91, 126
 - GPRS 156–9
 - HTTP 344–7, 356–61
 - IP 165–75, 180–3, 193–6
 - IrDA 126, 129, 404
 - OBEX 274–87, 289, 310–12, 320–33, 339–41
 - progress notifications 47–8, 193–6
 - sockets 33–8, 53–62, 84–91, 104–6, 165–75, 180–3, 193–6, 237–40, 345–7
 - subconnections 49–50, 159–60, 198–203
 - TCP socket 180–3, 358
- Connection ID headers, OBEX 278–9, 289, 321–2, 333, 335–41
- connection management 8–9, 27, 28, 46–52, 159–60, 162–204
 - see also* RConnection
 - concepts 8–9, 27, 28, 46–52, 159–60, 162–204
 - monitoring services 46–50
 - starting/stopping services 46–50, 165–75, 182, 185–6, 193–6
 - subconnections 49–50, 158, 198–203
- connection-oriented protocols, concepts 31, 80–2, 93–4
- connectionless protocols
 - see also* UDP
 - concepts 31, 183–6
- connectivity
 - see also* communications
 - concepts 4–12, 46–50
 - multiple options 46–50
- ConnectL 310–12, 333–4, 336–7
- ConnectSocketL 145–8
- ConRef 376, 378, 386–7
- ConstructL 53–4
- controller, Bluetooth 17, 65–6, 82–3, 121–4
- ConvertCharacterSetL 140
- cookies 21, 274, 344, 356, 364
- Copy command, OMA Device Management system 385, 387
- CopyL 294, 300
- costs, messaging 243
- CPhoneEngine 208–14
- CPMs (comms provider modules), concepts 15
- CPP files 52
- CProgressTracker 196
- CQBTUISelectDialog 86–7
- CQikSendAsDialog 243–4
- CQikSendAsLogic 243–4, 254–5
- CreateL 123–4
- CreateServiceRecordL 104–5
- CreateSummaryGeneratorsL 222, 227–32
- CreateView 110–11
- CRemConCoreApiController 122–4
- CRemConCoreApiTarget 122–4
- CRemConInterfaceSelector 122–4
- CRichText 252–4
- cryptography 6
- CS60BtDeviceDiscoverer 87–8
- CSD 167–9, 205
 - see also* circuit-switched networks
- CSdapEngine 115–18
- CSdpAgent 106–7, 113–18
 - see also* RNetDatabase
 - concepts 52, 106–7
- CSdpAttrIdMatchList 109, 116–18
- CSdpAttrValue... 107–9, 112–18
- CSDPHelper 114–18
- CSdpSearchPattern 107–9
- CSecureSocket 186–93
- CSender 249–50
- CSendUi 243–4, 251–3
- CSendWorkbenchAppUi 252–3
- CSmlDmAdapter 379–87
- CSmsBuffer 238–40
- CSmsMessage 225–6, 238–40
- CSocket, concepts 56–62
- CSocketFountain 56–62
- CSocketReader 60–2
- CSocketWriter 59–62
- CSR (Cambridge Silicon Radio) 393
- CSubConGenEventDataClient-Joined 202–3
- CSubConGenEventDataClient-Left 202–3
- CSubConGenEventParams-Changed 202–3
- CSubConGenEventParams-Granted 202–3
- CSubConGenEventSubConDown 202–3
- CSubConParameterFamily 199–203
- CSubConQosGenericParamSet 199–203
- CSubConQosR99ParamSet 199–203
- CSummaryScreenUI 218–21
- CSYName 402–4
- CSYs (Comms SYstem plug-ins), concepts 14, 17, 19, 402–4
- CTelephony 208–14
- CTS 79, 129–30

- CurrentCipherSuite 186–93
- CWeatherReportWatcher
 - 218–21, 235–40
- CWebPage 54–5
- data access, telephony 205–6, 211–12
- data caging 248–50
- data element alternatives (DEA) 77, 109
- data element sequences (DES) 77, 104–5, 109
- data synchronization 11, 21, 23, 343, 365–87
- datagram sockets
 - see also* sockets
 - concepts 31, 40, 55–6, 60, 120, 128–9, 145–6, 148–9, 183–6, 237–40
- DataReceivedNotification-Request 49
- DataSentNotification-Request 49
- DataTransferredRequest 49
- Date headers, OBEX 276–7
- DDF (device description framework), OMA Device Management system 368–70, 379–87
- DDFStructureL 381–7
- DEA (data element alternatives) 77, 109
- debugging 5, 43, 248–9, 262–3
- default/implicit connections, Symbian OS 163, 246
- Delete command, OMA Device Management system 385
- DeleteMasked 299–300
- DeleteRecord... 105
- DeregisterServiceL 144
- DES (data element sequences) 77, 104–5, 109
- descriptors 6, 40, 53, 59, 133–6, 238–9
 - see also* HBufC; RBuf; TBuf...; TDes...; TPtr...
 - sockets 40
- design considerations, messaging 233–5
- DevDetail 387
- developers 3–12, 15, 52, 244–5, 391–406
 - background 3–12
 - emulator 7–9, 15, 52, 244–5, 391–406
 - practical information 8
 - types 3–4
- device class
 - Bluetooth 68–9, 88–91, 112–18, 288, 318–22
 - CoD 68, 88–91, 112–18, 288, 318
 - IrDA 138–40
 - USB concept 64
- device description framework (DDF), OMA Device Management system 368–70, 379–87
- device discovery
 - Bluetooth 66–9, 83–91, 112–18, 287–8, 305–8, 318–22, 409–10
 - IrDA 125–9, 130–40, 145–6, 288, 305–8, 317–22, 336
- Device Management system (DM) 4, 8, 9, 14, 21, 22–3, 365–87, 392
- adapters 367–87
- commands 384–7
- concepts 22–3, 365–87, 392
- definition 365–6
- DFProperties 373–87
- error values 382–4
- essentials 367–72
- high-level overview 366–7
- device names, concepts 50–2, 132–3, 139–40
- DevInfo 387
- DFProperties 373–87
- DHCP 19, 167–9, 400–1
- Dial 212–13
- dial-up connections 106, 164
- DialCompleteL 209
- DialogMode 186, 192
- dialogs, security issues 98–100
- DialOutISP 402–3
- DialupNetworkingUUID 106
- Diffie–Hellman cipher suites 189
- directed connections, OBEX 278–9, 321–2, 333–42
- directories, OBEX SetPath command 296, 313–14, 330–2
- Disconnect 314–15
- Disconnect command, OBEX 274–5, 279–80, 284, 289, 314–15, 322, 339–40
- disconnecting
 - OBEX 274–5, 279–80, 284, 288, 289, 314–15, 322, 339–40
 - sockets 42–5, 62, 97, 111, 152–3, 173–5, 182, 194–6
- discoverability concept
 - Bluetooth 66–9, 83–91, 112–18, 287–8, 305–8, 318–22, 336, 409–10
 - IrDA 125–9, 130–40, 145–6, 288, 305–8, 317–22, 336
- DiscoverDeviceAndService 145–6
- DiscoverDevicesL 85–91, 130–40
- DiscoverServicesL 137–40
- DLLs 16, 113, 255–7, 268–9, 290, 338
 - see also* message-type modules
- DM *see* Device Management system
- DNL (Direct Navigational Link) 213–14
- DNS (domain name system) 18, 31, 50–2, 54, 161, 168–9, 175–6, 177–80, 189–93
 - error codes 178–80
 - IP 161, 168–9, 175–6, 177–80, 189–93
- DoCoMo 11
- domain name system (DNS) 18, 31, 50–2, 54, 161, 168–9, 175–6, 177–80, 189–93
- DomainName 375
- double-buffering strategies 55, 62

- downloads
 - code 9
 - managers 343
- DRM 210, 363
- DSR 129
- DTDs 370
- DTR 129, 404
- DTS 129
- DUN 106

- E-series, Nokia 410
- EAlreadyExists 382
- EAlways 337–8
- EAnyTransactionEvent 363–4
- ECancel 348–50
- Ecom 362–3, 367, 379–81
- ECommDbBearer... 171
- ECommDbDialogPref... 170–2
- ECommitFailed 383
- ECommitOk 383
- EConnectingSocket 53–4, 145–6
- ECUART 402–3
- EDGE 28, 205
- EDiscoveringDeviceAndService 145–6
- EDiskFull 382
- EditL 262–3
- editors, messaging 243, 248, 262–3
- EDns... 179–80
- EError 382
- EFailed 351–3
- EGotResponseBodyData 351–2
- EGotResponseHeaders 350–2
- EGotResponseTrailer-Headers 351–2
- EHttpPipelining 359–60
- EHttpSocketConnection 346–7, 360–1
- EHttpSocketServ 346–7, 360–1
- EIASDataMissing 134–5, 138–40
- EIfPresent 337–8
- EImmediate... 42–6, 182
- EInvalidObject 382
- EIr... 153

- EListening 147–8
- emails 8, 10, 20, 50, 217–20, 223–32, 242–71
 - message store structure 223–6
 - MIME headers 225, 226, 230
 - POP3 10, 20, 218, 220, 223–32, 246, 392
 - SendAs 242–71
 - server 223–4
 - services concepts 246
 - SMTP 10, 20, 50, 220, 245–6, 392
 - summary screens 218–21
- EMaxNumTransactionstoPipe-line 359–60
- EMaxNumTransportHandlers 359–60
- emergency calls 213
- EMsvEntriesChanged 231–2
- EMsvEntriesCreated 232
- EMsvEntriesDeleted 232
- EMsvServerReady 222
- EMsvSortByDateReverse 227–9
- EMtudCommandSendAs 262
- emulator 7–9, 15, 52, 244–5, 391–406
 - Bluetooth 391–4, 403–6
 - concepts 52, 391–406
 - examples 52
 - IP 396–403
 - IrDA 391, 394–6, 403–6
 - MTMs 244–5
 - S60 392–4, 395–403
 - serial ports 391–6, 404–6
 - telephony 391, 403–6
 - UIQ 392–3, 395–403
- encoding
 - HTTP 344, 363
 - OBEX headers 275–6, 281
- encryption 6, 71–2, 97–103, 186–93, 319
- End-of-Body headers, OBEX 276–7, 281
- engines
 - see also* active objects
 - concepts 208–9
- ENoChecking 337–8
- ENoMemory 383

- ENormal 42–6
- ENotFound 382
- Entry... 229–30
- EnumerateConnections 48
- EnumerateSubConnections 49
- EObjectInUse 383
- EOk 382
- EPOC 15, 28, 289, 398, 404–6
- EPOST 354–5
- Eproxy... 355
- EReceivingData 53–62
- ERedirectedPermanently 351–2
- ERedirectedRequiresConfirmation 351–2
- ERedirectedTemporarily 351–2
- ERegisteringService 147–8
- ERemConCoreApiButtonClick 122–4
- ERemConCoreApiButtonPress 122–4
- ERemConCoreApiButtonRelease 122–4
- EResponseComplete 351–3
- Ericsson 10–11, 409, 410
- ERollbackFailed 382
- ERollbackOk 383
- err 324–32
- Error 202
- error codes 8, 111, 132, 171, 173–5, 178–80, 195, 315–17, 323–4, 330–3, 382–4
 - Bluetooth 111
 - DNS 178–80
 - IP 171, 173–5, 178–9, 195
 - IrDA 132
 - OBEX 315–17, 323–4, 330–3
- ErrorIndication 332–3, 339–40
- ESendAsRecipientTo 247–8
- ESendingData 53–62
- ESOCK 9, 10–12, 14–19, 27–62, 130–53, 161, 181, 205–6, 345
 - see also* RConnection;
 - RSocket; sockets
- access methods 29–46

- ESOCK (*continued*)
 - addresses 50–2, 54, 86
 - Bluetooth 83
 - concepts 10–12, 14–15, 27–62, 130–53, 161, 181, 205–14, 345
 - connection management 8–9, 27, 28, 46–50, 181
 - examples 52–62
 - generic socket options 41–4, 133, 236–7
 - historical background 10–12, 28
 - HTTP 345
 - introduction 27–62
 - IrDA 31–3, 130–53
 - naming and lookup services 50–2, 54, 183–4
 - overview 27–52
 - roles 27–52, 130–1
 - sockets API overview 27–8
 - telephony 205–14
- esock 16, 52, 56
- es_sock.h 46–7, 130, 144
- EStopInput 42–5
- EStopOutput 42–5
- ESubmit 362
- ESucceeded 351–3
- Etel 11, 14, 19, 205–14
 - see also* telephony
 - 3rd party 19
 - concepts 14, 19, 205–14
 - functionality 211–12
 - historical background 211
 - uses 206–7
- Ethernet 16–17, 50, 63–4, 164–5, 397, 399–402
 - see also* WLAN
- ETooLargeObject 382
- EventNotification 202–3
- events 55–62, 201–3, 350–3, 363–4
 - see also* active objects
- example applications, background 9
- exclusions, technologies 5–6
- EXE files 269
- Execute command, OMA Device Management system 385, 387
- explicitly-loading filters 362–3
- EXPORT_C 59
- Extended Systems Jeteye 395–6
- fax 10
- FetchLeafObjectL 384
- files
 - attachments 248–50, 252–3, 254–5, 263–5
 - data caging 248–50
 - OBEX 276–8, 286–7, 289, 290–304, 308–16, 322–6
- filters
 - device classes 90
 - HTTP 22–3, 344–5, 347–9, 357–8, 361–4
- Find... 169–70, 298, 315, 325
- FindProtocol 29–30
- firewalls, HTTP 343
- firmware 365–6
- First 298–9, 325, 328–9
- Flickr 241, 257–71, 344, 346, 392, 415–17, 419–20
 - client MTM 263–5
 - concepts 258–71, 344, 392, 415–17
 - data MTM 259, 260–2
 - HTTP 344, 346
 - server MTM 264–9
 - UI MTM 256, 259, 262–3
- FlickRestOperation-Complete 266–8
- FlickrMTM 241, 257–71, 415–17, 419–20
 - classes 258–9
 - concepts 258–71
 - shared settings 269
- FlushUpload.dll 266
- FlushSessionCache 187, 192
- folder entries, message store structure 223–4
- FTP 50, 90, 273, 336–7
- Fujitsu 11, 12
- functionality issues, user interfaces 4
- functiontext 262
- future prospects 407–11
 - interactions 407, 409–10
 - networks 407–10
 - services 407, 410–11
 - gaming 112–18, 160–1, 217
- GAP 83–124, 288
- Gateway Node (GN) 164
- GAVDP (generic audio video distribution profile) 65, 83, 119–24
- generic attributes, message entries 224–5, 229–31
- generic implementations, concepts 3–4, 133
- Generic Object Exchange Profile (GOEP) 273–4
- generic socket options, ESOCK 41–4, 133, 236–7
- GenericNetworkingUUID 106
- geographic factors, IP 161
- GERAN 2.5G network 156
- Get... 29–30, 34–6, 41–2, 48–50, 51–2, 123–4, 131–2, 133–4, 176–7, 178–80, 211–13, 312–16
- Get 312–16
- Get command
 - HTTP 343–4
 - OBEX 274–5, 279–80, 285–6, 289, 292–3, 312–16, 326–30, 333, 339–41
 - OMA Device Management system 384–5
- GetBodyTextL 225–6
- GetByAddress 52, 86, 179–80
- GetByName 51–2, 131–2, 176–7, 178–80, 312–13
- GetCall... 212–13
- GetCompleteIndication 327, 329–30, 339–41
- GetConnectionInfo 48–50
- GetCredentials 359–61
- GetCurrentNetwork... 212
- GetDesSetting 196
- GetEntry 229–30
- GetField 350, 356
- GetFlightMode 211–12
- GetHeaderCollection 350
- GetIntSetting 196
- GetLuidAllocL 386

- GetNextBodyPart 351
- GetNextDataPart 349, 352–3
- GetOperatorName 212
- GetOpt 41–2, 182–3, 401
- GetPacketIndication 327, 329–30, 339–41
- GetPage 52–4
- GetParam 350, 356
- GetParameters 49
- GetPhoneId... 210–12
- GetProtocolInfo 29–30, 34–6
- GetPutFinalResponsibilities 315–16
- GetRawField 350
- GetRemoteDevAddr 133–40
- GetRequestIndication 326–30, 339–41
- GetSubConnectionInfo 49–50
- GetSubscriberId... 210
- GETting concepts 274, 292
- GetUserPasswordL 334–5
- GGSN (general GPRS support node) 156–9, 161, 199
- GlobalSettings 404
- GN (Gateway Node) 164
- GOEP (Generic Object Exchange Profile) 273–4
- GPRS 11, 13–14, 28, 155–65, 167–9, 174–5, 194–6, 211, 357–8, 403–4
 - see also* IP
 - APNs 156–7, 162–3, 167–9
 - billing practices 157
 - concepts 155–65, 167–9, 174–5, 194–6, 211, 357–8, 403–4
 - connecting 156–9
 - limitations 158–9
 - overview 156–9
 - passwords 157, 167–9
 - PDP contexts 157–8, 197–8, 203
 - primary contexts 158, 198–9
 - QoS 156, 158
 - secondary contexts 156, 158, 198, 203
 - usernames 157, 167–9
- GPS 69
- GSM 13–14, 79
 - see also* circuit-switched networks
- GZip content encoding filters 363
- H4 protocol 391–4
- Handle... 103, 221–2, 231–2, 235
- handles, connections 163
- HandleSessionEventL 221–2, 231–2, 235
- handshakes, secure sockets 186–93
- HBuFC 40, 329
- HCI (host controller interface) 17, 65–6, 83, 111, 392–4
- HDLC protocol 79
- headers
 - emails 225, 226, 230
 - HTTP 277, 345–6, 350–3, 364
 - OBEX 274, 275–81, 282–7, 289–304, 309–16, 321–33, 335–41
- heap-allocated classes 6, 102, 293–4
- hint bits, IrDA 142–3, 150–1
- hold mode, physical links 71–2, 92–3
- host
 - Bluetooth 17, 65–6, 82–3
 - IrDA 131–2
- host controller interface (HCI) 17, 65–6, 83, 111, 392–4
- host resolvers, concepts 50–2, 84–91, 111, 131–2, 160, 162–3, 174–5
- HSDPA 49
- HTTP 4–11, 14, 21–3, 39, 50, 52–62, 128, 181, 241, 273–4, 277, 280, 292, 341, 343–64, 392
 - APIs 343–64
 - architecture 344–5
 - authentication 344, 359–62
 - body data 348–53
 - cancellations 348–9, 353
 - classes 344–64
 - client-server framework 343–64
 - closing transactions 353
 - concepts 21–3, 50, 52, 181, 241, 273–4, 277, 280, 292, 341, 343–64, 392
 - connecting 344–7, 356–61
 - connection management 356–61
 - cookies 21, 274, 344, 356, 364
 - creating a session 344–7, 353–5
 - definition 343
 - event codes 350–3
 - examples 52–62, 241, 343–7
 - filters 22–3, 344–5, 347–9, 357–8, 361–4
 - firewalls 343
 - Flickr 344, 346
 - framework 343–64
 - Get command 343–4
 - headers 277, 345–6, 350–3, 364
 - monitoring transactions 350–3
 - multihoming 360–1
 - OBEX 21, 22, 273–4, 277, 280, 292, 341
 - passwords 359–61
 - performance issues 356–61
 - persistent connections 357–61
 - pipelining 22, 347, 357–61
 - Post command 343–4, 349
 - proxy support 355
 - receiving body data 352–3
 - receiving headers 350–3
 - request methods 343–64
 - response codes 39, 181, 280
 - response methods 343–64
 - roles 343–4
 - security issues 344, 348–9, 359–62
 - sessions 344–7
 - stateless factors 274, 344–5
 - status codes 280
 - Stringpool 344–5, 353–5
 - supplied body data 348–9
 - support 22
 - Symbian OS 343–64
 - TCP binding 274, 343, 345, 358
 - transactions 345–53, 361–4
 - uses 343–4
- HTTP/1.0 22, 344

- HTTP/1.1 22, 343–4, 357
- https 344
- IACs (inquiry access codes) 67
- IANA registered media types 277
- IAPs (Internet Access Points)
 - 162–204, 355, 360–1, 401–2
 - see also* IP
- IAS (Information Access Service)
 - 127–9, 133–44, 146–7, 278, 288, 306–7, 318
- ICC 208–210, 214
 - see also* SIM cards
- ICMP 33, 95–6, 161, 180
- IDEs (integrated development environments) 7
- IETF 64, 343
- IfAuth... 402–3
- IfName 402–3
- iFoundry pods 395
- IMAP4 10–11, 20, 220, 223–4, 246, 255–6, 392
- IMEIs (phone IDs) 206, 207, 209–11
- implicitly-loading filters 362–3
- IMPORT_C 84–5, 100, 102, 104, 106, 133–4, 140, 186–7, 345–50, 353–4
- IMS 155
- IMSI numbers 206, 207–8, 209–11
 - see also* SIM cards
- inboxes 228–9, 235, 263–5
- incoming calls, telephony 207–9, 213–14
- IncomingConnection 56–7
- INET_ADDR 177
- Info 34–6
- Information Access Service (IAS)
 - 127–9, 133–44, 146–7, 278, 288, 306–7, 318
- information sources 9–10
 - Symbian OS Library 9–10, 211, 372–3
 - websites 7, 9–10, 413–14
- infrared *see* IrDA
- Infrared Data Association *see* IrDA
- Input 176–7
- inquiry access codes (IACs) 67
- InstallMtmGroup 269–70
- integrated development
 - environments (IDEs) 7
- interactions, future prospects 407, 409–10
- intercepted SMSs 218, 235–40
- Interested 300
- interfaces
 - see also* user interfaces
 - M (abstract interface) classes 6, 299–300
- internal tables, Bluetooth registry 109–11
- internals, Symbian OS 14–15
- Internet 10–11, 22, 31, 54–6, 155–204, 343–64, 386–7, 401–2, 413
- Internet Access Points (IAPs)
 - 162–204, 355, 360–1, 401–2
 - see also* IP
- Internet Engineering Task Force 413
- introduction 3–12
- InvokeAsyncFunctionL 264–5
- Ioctl 39, 41–2, 96, 151–3, 166, 181–3, 237–8, 401–2, 404
- IOCTL_SERIAL... 404–6
- IP 8, 11, 14, 18–19, 28, 30–3, 47, 49–50, 155–204, 205–6, 347, 355, 360–1, 396–404, 406, 409–10
 - see also* GPRS
- active connection information
 - 196
- addresses 31, 50, 54–6, 147–8, 159, 176–80, 185–6
- bearer technologies 160–2, 163–75, 193–6, 392, 410
- Bluetooth 155, 160–1, 164–5, 409–10
- classes 162–3
- CommsDat 47, 161–3, 167–70, 174–80, 196, 205–6, 347, 355, 360–1, 397–9, 402–4, 406
- communications options 28, 49–50, 155, 158, 160–1
- concepts 18–19, 28, 30–1, 49–50, 155–204, 396–403, 409–10
- connecting 165–75, 180–3, 193–6
- connection parameters 167–71
- connection preferences 170–1
- default/implicit connections 163, 246
- different bearers 172–3, 193–6, 391–2, 410
- disconnections 173–5, 182, 194–6
- DNS 161, 168–9, 175–6, 177–80, 189–93
- emulator 396–403
- error codes 171, 173–5, 178–9, 195
- geographic factors 161
- IAPs 162–204, 355, 360–1, 401–2
- inactivity measures 172–3, 174–5
- information gathering 193–7
- introduction 155
- multihoming 159–60, 162–3, 360–1
- multiple IAPs 172
- passwords 157, 167–9
- per-technology bearer tables 167–9
- performance issues 164–5, 197–203
- progress notifications 47–8, 193–6
- QoS 156, 158, 197–203
- receiving data 175–93
- roles 155, 160–1, 409–10
- S60 158–9, 163, 164, 166, 174, 396–7
- secure sockets 12, 22, 186–93
- sending data 175–6, 180–93
- starting the connection 171–3, 193–6
- stopping the connection 173–5, 182, 185–6, 194–6
- Symbian OS 159, 160–204, 396–403
- TCP 158, 176–7, 180–3
- timers 172–3, 174–5
- UDP 158, 161, 172, 176, 180–1, 183–6

- UIQ 158–9, 163, 166, 174, 396–7
- usernames 157, 167–9
- uses 155, 160–1, 409–10
- using the connection 175–93
- IPSec 6
- IPv4/IPv6 stack 11, 18–19, 31–3, 176–7
- IR pods 8
- IrCOMM 18, 127–30, 402–3
- IrDA (Infrared Data Association)
 - 4–10, 14–22, 27–8, 31, 33, 50, 53, 125–53, 243, 245–6, 273–4, 278, 282, 288–9, 305–10, 317–22, 336, 338, 391–2, 394–6, 403–6, 413
 - see also* infrared
 - advantages 125–6
 - API definitions 130
 - Bluetooth contrasts 126
 - client sockets 144–6
 - concepts 4–10, 14–22, 31, 33, 50, 53, 125–53, 243, 245–6, 273–4, 278, 282, 288–9, 305–10, 317–22, 336, 391–2, 394–6, 403–6, 413
 - connecting 126, 129, 404
 - deadlock problems 128–9
 - device discovery 125–9, 130–40, 145–6, 288, 305–8, 317–22
 - device names 132–3, 139–40
 - device nicknames 132–3, 139–40
 - discoverability concept 125–9, 130–40, 145–6, 288, 305–8, 317–22, 336
 - emulator 391, 394–6, 403–6
 - error codes 132
 - hardware availability 126
 - hint bits 142–3, 150–1
 - historical background 125–6, 273–4
 - IAS 127–9, 133–44, 146–7, 278, 288, 306–7, 318
 - inbound data transfers 144, 146–8
 - line-of-sight requirements 125–6, 278, 288, 305–8
 - monitored link status 153
 - OBEX 18, 20, 21–2, 128, 137–40, 142–3, 245–6, 256, 273–4, 278, 282, 288–9, 305–10, 317–22, 336, 392
 - outbound data transfers 144–6
 - overview 125–9
 - P&S 153
 - Pods 394–6
 - protocols 31–3, 130, 144–8, 273–4, 306–7
 - range statistics 126
 - reading/writing methods 148–9
 - security issues 125, 128, 278, 338
 - server sockets 144, 146–8
 - service discovery 125–9, 133–40, 145–6, 288, 305–8, 317–22, 336
 - service registration 140–3, 147
 - service removal 142, 143–4
 - socket options 149–53
 - speed statistics 126
 - stack 127–9
 - standards 125–9
 - suppressed discovery responses 132–3, 149–51
 - Symbian OS implementation 129–53, 394–6
 - TinyTP 18, 31, 33, 127–32, 145–6, 148–51, 274, 288, 306–7, 310
 - visible devices 132–3
- IrDIAL 129
- ir.irda.esk 396
- IrLAP 127–9, 149–50, 152–3
- IrLMP 127–9, 288
- IrMC 286, 330
- IrMUX 31, 33, 127–9, 131–2, 145–6, 151
- IrNET 129–30
- irobex.dll 290
- IrPHY 127–9
- ir_sock.h 130, 146–7
- iSIREnable flag 130
- IsList 134–40
- iSmallText 254
- ISO character set encoding 135–6
- isochronous data 72–4
- ISPs 161–4, 168–9, 402–3
- IssueAcceptL 58
- IssueImagePostL 266–7, 349
- Java 3–4, 343
- jitter performance measure 61, 198–203
- JPG images 249–50, 276
- KAfInet... 33, 53–4, 176–7, 183–6
- KAutoBindLSAP 146–8
- KAVCTP 33
- KAVDTP 33
- KBTAddrFamily 33
- KBTLinkManager 33
- KBTSecurityDeviceOverride 102
- KCDTIdIAPRecord 168–70
- KCharacterSet-Identifier... 135–40
- KCommsDbSvrRealMaxFieldLength 196
- KConnectionFailure 195–6
- KConnectionStartingClose 195–6
- KConnectionTypeDefault 46–7
- KConnectionUninitialised 193–6
- KDataTransferTemporarilyBlocked 194–6
- KDiscoveredIndicationIoctl 152
- KDiscoveryIndicationIoctl 151, 153
- KDiscoveryResponseDisableOpt 133
- KDiscoverySlotOpt 149–51
- KErrAbort 314
- KErrAccessDenied 175, 305
- KErrAlreadyExists 142
- KErrArgument 317
- KErrCancel 42, 173–4

- KErrConnectionTerminated 173–4
- KErrCorrupt 135
- KErrDisconnected 97
- KErrEof 178–80
- KErrHCILinkDisconnected 97
- KErrHCILinkDisconnection 111
- KErrInUse 317
- KErrIrObexRespTimedOut 341
- KErrL2CAPAccessRequest-Denied 111
- KErrNone 34–6, 59, 87, 131, 138–40, 171, 195–6, 248, 260–1, 298–9, 314–15, 317, 324–5, 328, 330–1
- KErrNotFound 317
- KErrNotReady 173–5, 191–3
- KErrNotSupported 102–3, 135, 186–7, 260, 317
- KErrPermissionDenied 174
- KErrReflexiveBluetooth-Link 111
- KErrServerBusy 29
- KErrSSLSocketBusy 189
- KErrTimedOut 317
- KErrTooBig 94–5
- KErrUnsupported 134–5
- KErrWouldBlock 41
- KExclusiveModeIoctl 151
- KExpeditedDataOpt 149–50
- KFinishedSelection 194–6
- KFirstHintByteOpt 150–1
- KGIAC 84–5
- KHCIErrorBase 111
- KHostMaxDataSizeOpt 148–9, 150
- KHostMaxTATimeOpt 150
- KIASClassNameMax 134
- KIdleClearRequestIoctl 152
- KIdleRequestIoctl 151–2
- KInetAddr... 176–7, 184–6
- KIoctlSelect 42
- KIoctlTcpNotifyDataSent 181–3
- KIrdaAddrFamily 33, 130, 144–8
- KIrdaPropertyCategory 153
- KIrdaStatus 153
- KIrlap... 150, 151–2
- KIrlapResetIndication-Ioctl 152
- KIrlapResetRequestIoctl 152
- KIrmux... 33, 130, 145–6, 151–2
- KIrResponseCharSet-ConversionPanic 136–7
- KIrTinyTP 33, 130, 145–6
- KL2CAP 33
- KLIAC 84–5
- KLinkLayerClosed 195–6
- KLinkLayerOpen 171–2, 194–6
- KLocalBusyCleared 150
- KLocalBusyDetected 150
- KMaxSummaryMessages 229–30
- KMTMStandardFunctionsSendMessage 264, 265–6
- KMultipleModeIoctl 151
- knowledge levels 6–7
- KNullDesC 254
- KObexHdrName 292
- KObexHdrType 292
- KObexIrTTPProtocol 307
- KObexIrTTPProtocolV2 306–7
- KPortNum 177
- KPrivateAttachmentFile 253, 255
- KProtocolInet... 33, 53–4, 183–6, 203
- KRemoteMaxDataSizeOpt 148–50
- KRequestPending 310–11
- KRFCOMM 33
- KS0... 43–4
- KSecondHintByteOpt 150–1
- KSI... 35–8, 40, 42–4
- KSmallText 254
- KSocketDatagram 33, 145–6, 183–6, 237–9
- KSocketBufSizeUndefined 43
- KSocketRaw 33
- KSocketReadContinuation 30–1, 40–1, 60–1, 148–9
- KSocketSelect... 43–4, 237–40
- KSocketSeqPacket 33, 145–6
- KSocketStream 33, 53–4, 203
- KSocketUrgentWrite 38–9
- KSolBtL2CAP 102
- KSolBtSAPBase 91
- KSOLSocket 41–2
- KSoTcpAsync2MslWait 182
- KSoTcpKeepAlive 182–3
- KSoTcpLinger 182
- KSoTcpReadBytesPending 183
- KSoTcpSendBytesPending 182–3
- KSoUdpSynchronousSend 166, 184–6
- KStartingSelection 194–6
- KTinyTPDisabledSegmentation 149, 151
- KTinyTPLocalSegSizeOpt 150
- KTinyTPRemoteSegSizeOpt 150
- KUIDMscMessageEntry 262–3
- KUIDMsgTypeFlickr 261
- KUIDMtmQuery... 260–71
- KUnexpeditedDataOpt 149–50
- KUserBaudOpt 150
- L2CAP 16, 31, 33, 61, 64–6, 72–4, 78–84, 93–6, 102–3, 105–6, 111, 114–24
 - see also* asynchronous transport concepts 72–4, 78–84, 93–6, 102–3, 105–6, 111, 114–24
 - critique 82
 - error-control feature 74, 78–82
 - isochronous data 73–4
 - per-channel flow control 74
 - ports 80–2
 - RFCOMM uses 81–2, 96, 116–18
 - sockets 93–5, 102–3
 - version 1.2 73–4, 82
- Language 376
- laptops 88–9, 110–11, 125, 393
 - device class 88–9
 - IrDA 125–6
- LastProgressError 175, 195–6
- LastServerResponseCode 341

- LastSessionClosedTimeout 174–5
- LastSocketActivity 174–5
- LastSocketClosedTimeout 174–5
- latency performance 61, 71–4, 197–203, 357–8
- LaunchEditorAndCloseL 248
- LeaveIfError 37, 53–4, 58, 85, 90–1, 110–11, 131–2, 138–41, 145–8, 171–2, 178–9, 237, 247–50, 255, 297, 346
- Length headers, OBEX 276–7, 280–1, 299–300
- licensees, flexibility 4–5
- lifecycles, sockets 31–46
- line-of-sight requirements, IrDA 125–6, 278, 288, 305–8
- Link Manager 33
- linked attachments 249–50
- Listen 32, 36–7, 56–62
- listening 32, 36–7, 56–62, 146–8, 153, 180–3
- _LIT_SECURITY_POLICY... 45–6
- LM-IAS 127–9
- LM-MUX 127–9, 149, 152
- local unique identifiers (LUIDs) 381–6
- LocalAddr 183–6
- LocalName 147–8, 185–6
- LocalServices 21, 89–91, 100, 149–51, 268–9, 338
- Location 212
- logical structure, SDDBs 75–6
- logical transports, Bluetooth 66, 71–4, 91, 114–18
- loss performance measure 61, 198–203
- LSAP_SELs 127–9, 137–40, 146–8, 152, 288
- LUIDs (local unique identifiers) 381–6
- M (abstract interface) classes 6, 299–300, 334
 - see also mixin classes
- MAC addresses 399–400
- management objects (MOs), OMA Device Management system 368–87
- management trees, OMA Device Management system 368–87
- master devices, slaves 69–72, 112–18
- maximum transmission units (MTUs) 94–5, 262, 308–16, 400–1
- MBluetoothBusObserver 114–18
- MBluetoothSocketNotifier 103, 114–18
- MCSocketCallbacks 56–7, 59–60
- message, definition 348
- message entries 223–32, 245–6, 248–50, 263–5
 - see also emails; SMS
- APIs 226–32
- application-specific SMSs 235–40
- attachment entries 223–6, 245–6, 248–50, 263–5
- CMsvEntry 226–35
- generic attributes 224–5, 229–31
- message store structure 223–6
- specific data 225–6, 230–2
- structure 224–6, 263–5
- TMsvEntry 225–6, 229–31
- message store
 - changes 234–5
 - concepts 20–1, 217, 220–1, 223–40, 246–50, 256–7, 263–5
 - data caging 248–50
 - entry types 223–4
 - structure 223–4, 263–5
- Message Suite 10
- message-type modules (MTMs) 9, 10–11, 218, 220–2, 230–1, 242, 243–71, 415–17
 - see also Flickr...; messaging
- basic capabilities 245–6, 260–2
- brief background 255–6
- classes 258–9
- concepts 20–1, 218, 220–2, 230–1, 242, 243–71
- diagram 257
- drawbacks 257
- historical background 255–6
- installation 269–71
- registration files 269–71
- security issues 268–9
- services/accounts 246
- signed software 269
- tests 258
- transport segmentation 256–7
- types 20–1, 244–6
- messaging 4, 7–8, 9, 10–11, 14–15, 20–1, 112–18, 217–40, 241–71, 392
 - see also MMS; SendAs; SMS
- abstract concepts 255–6
- asynchronous behaviour 233–4, 236–7, 248, 267
- attachments 223–6, 245–6, 248–50, 252–3, 254–5, 263–5
- concepts 14–15, 20–1, 217–40, 241–71, 392
- costs 243
- data caging 248–50
- design considerations 233–5
- drawbacks 257
- editors 243, 248, 262–3
- example applications 218–20, 241–71
- FlickrMTM 241, 257–71, 415–17, 419–20
- high-level sending overview 242–5
- introduction 217–18, 241–2
- key concepts 217–18, 241–5
- performance issues 233–5
- push messages 223–4, 241–2
- receiving messages 217–40, 246
- S60 242–71
- security issues 210, 222, 268–9
- Send dialogs 243–4, 250–71
- sending messages 241–71
- server functions 217, 220–40, 245–50, 256–7, 265–8
- server sessions 220–2
- summary screens 217–35

- messaging (*continued*)
 - Symbian OS 217–40, 241–71, 392
 - third-party application types 217–18
 - UIQ 242–71
- metadata 276
- MHFLoad 363–4
- MHFRunL 348–9, 364
- MHTTPAuthenticationCallback 359–63
- MHTTpDataSupplier 349
- MHTTTPFilter 363
- MHTTTPTransactionCallback 348–9
- Microsoft Windows 132–3, 391–406
- MIdentifiableSocketNotifier 114–18
- MIME headers 225, 226, 230, 369–77
- MIncomingConnection 56–7
- Mitsubishi 12
- mixin classes
 - see also* M (abstract interface) classes
 - concepts 299–300, 334
- MMP files 13
- MMS (multimedia messaging service) 7–8, 11, 20, 159–61, 163–5, 167, 217, 220, 223–4, 229, 243, 245–6, 256, 392, 396, 404, 410
 - see also* messaging
- MMsvAttachmentManager 226, 264
- MMsvSessionObserver 220–2, 226–33
- MM.TSY 403–4
- MNotify 208–9
- MOBexAuthChallengeHandler 334–5
- MOBexHeaderCheck 293–304, 315
- MOBexServerNotify 301, 316–33, 338–41
- MOBexServerNotifyAsync 339–41
- mobile phones
 - device class 88–9
 - future prospects 407–11
 - geographic factors 161
- ModemBearer 402–4, 406
- ModemForPhoneServicesAndSms 404
- monitoring services
 - connection management 46–50
 - HTTP 350–3
- MOs (management objects), OMA Device Management system 368–87
- Motorola 10–11
- MRemConCoreApiControllerObserver 122–4
- MRemConCoreApiTargetObserver 122–4
- MSdpAgentNotifier 106–7, 115–18
- MSdpAttributeValueVisitor 116–18
- MSdpElementBuilder 108–9
- MSdpHelperNotify 114–18
- MSmldmAdapter 380–7
- MSmldmCallback 380–7
- MSmldmDDFObject 380–7
- MSmldmDdObject 381–7
- MTM_CAPABILITIES 269–70
- MTM_COMPONENT_V... 270–1
- MTM_INFO_FILE 269–70
- MTMs *see* message-type modules
- MTM_SECURITY_CAPABILITY_SET 269–70
- MTransactionCallback 348–9, 364
- MTUs (maximum transmission units) 94–5, 262, 308–16, 400–1
- multihoming 159–60, 162–3, 360–1
- multimedia framework 120–1
- multimedia messaging service (MMS) 7–8, 11, 20, 159–61, 163–5, 167, 217, 220, 223–4, 229, 243, 245–6, 256, 392, 396, 404, 410
- multimode ETel 206, 211–12
- multiplexing 70, 73–4, 79–82, 112–19, 125–9, 278, 289, 335–6
- MWeatherReportObserver 238–40
- Name 45–6
- Name headers, OBEX 276–7
- naming and lookup services
 - see also* addresses
 - concepts 50–2, 54, 183–4
- NAP accounts 373
- NETCON (network controller) 18–19
- network controller (NETCON) 18–19
- network interface manager (NIFMAN) 18–19
- network interfaces (NIFs), concepts 14, 18–19, 161, 168–9, 181, 193–6, 399–400
- NetworkControl 29, 89–91, 132–3, 150–3, 173–5, 180, 268–9
- networks, future prospects 407–10
- NetworkServices 22, 165–75, 211–12, 222, 268–9, 361
- NewL 40, 56–8, 122–4, 201–3, 295–308, 316–18
- NewLC 168–9
- Next 51–2, 132, 179–80, 298–9
- NextRecordRequestL 106–7
- nicknames, devices 132–3, 139–40
- NIFMAN (network interface manager) 18–19
- NIFs *see* network interfaces
- nifvar.h 193
- Nokia 7, 10–11, 401, 409, 414
 - 6600 11
 - 7650 11
 - 9100 11
 - 9210 11
 - E-series 410
 - N70 12
 - N72 12
 - N90 12
- non-blocking operations 38, 40–2

- notifications 47–8, 193–6, 209, 213
- NotifyChange 213
- NotifyNextBasebandChangeEvent 92–3
- NT RAS 397, 402–3
- NumItems 134–40
- NumProtocols 29–30

- obexclient.h 290
- ObexConnectIndication 321–2, 339–41
- obexconstants.h 291–2
- ObexDisconnectIndication 322, 339–40
- obexheaders.h 290, 295
- obexobjects.h 290
- obexserver.h 291
- ObexUnderlyingHeader 294–304
- Object EXchange (OBEX) 4, 7–11, 14, 20, 21–2, 65, 76, 83, 89–91, 105–6, 128, 137–43, 229, 245–6, 256, 273–342, 392
 - Abort command 274–5, 279–80, 286–7, 289, 314, 332, 339–41
 - API structure 290–342
 - asymmetric client-server protocol 274–5
 - asynchronous server API 338–41
 - authentication issues 275–8, 282–4, 289, 310–12, 333–42
 - bindings 274, 305–8, 316–20, 336
 - Bluetooth 20, 21–2, 65, 69, 76, 83, 89–91, 105–6, 245–6, 256, 273–4, 278, 286, 287–9, 305–8, 309, 318–22, 330, 336, 392
 - body concepts 274, 276–7, 281, 291–304, 309–16
 - buffer/file combinations 301–4, 308–16, 322–6
 - classes 290–1, 293–306, 316–33
 - client APIs 290, 304–16, 333–42
 - client-server protocol 274–9, 281–7, 290, 304–42
 - commands 274–5, 279–80, 281, 282–7, 289, 305–6, 310–16
 - concepts 4, 7–11, 14, 18, 20, 21–2, 83, 89–91, 105–6, 128, 137–40, 142–3, 229, 245–6, 256, 273–342, 392
 - Connect command 274–5, 279–80, 281, 282–7, 289, 310–12, 315, 321–2, 333, 335–6, 339–40
 - Connection ID headers 278–9, 289, 321–2, 333, 335–41
 - definition 273–5
 - directed connections 278–9, 321–2, 333–42
 - directories 296, 313–14
 - Disconnect command 274–5, 279–80, 284, 289, 314–15, 322, 339–40
 - encoding 275–6, 281
 - error codes 315–17, 323–4, 330–3
 - examples 275
 - failed connections 284
 - feature set 289–90
 - files 276–8, 286–7, 289, 290–304, 308–16, 322–6
 - fundamentals 274–81
 - Get command 274–5, 279–80, 285–6, 289, 292–3, 312–16, 326–30, 333, 339–41
 - headers 274, 275–81, 282–7, 289–304, 309–16, 321–33, 335–41
 - historical background 21–2, 273–4, 289
 - HTTP 21, 22, 273–4, 277, 280, 292, 341
 - immediate shutdown 314–15, 320, 333
 - introduction 273–4
 - IrDA 18, 20, 21–2, 128, 137–40, 142–3, 245–6, 256, 273–4, 278, 282, 288–9, 305–10, 336, 392
 - new-style API 293–304
 - object APIs 290–304
 - P&S 89–91
 - packet sizes 276–7, 280–1, 299–300, 308–10, 324
 - packets 274–81, 289–342
 - passwords 278, 282–7, 333–42
 - peers concepts 274–5, 281, 284, 333–42
 - performance issues 309–10, 329–30
 - Put command 274–5, 279–80, 284, 289, 292–3, 311–16, 322–6, 333, 339–41
 - request packets 274–8, 279–87, 305–16, 320–42
 - response codes 280, 282–7, 315–17, 323–5, 328, 330–2, 341
 - response packets 274–8, 279–87, 316–41
 - roles 273–5
 - security issues 275, 277–8, 282–4, 289, 310–12, 319–20, 333–42
 - server APIs 290, 315–42
 - service discovery 287–8, 305–16, 336
 - session protocol 274–8, 281–7, 321–2, 333–4
 - SetPath command 274–5, 279–80, 286, 289, 313–14, 330–2, 339–41
 - Symbian OS 273, 275, 289–342
 - Symbian OS v9.2 338–41
 - Target headers 278–9, 284, 289, 311, 321–2, 333, 335–41
 - timers 341
 - transports 287–8, 305–6, 316–22, 336
 - USB 21, 273–4, 289
 - uses 273–5
 - Who headers 278–9, 284, 289, 321–2, 333, 335–41
- object model, OBEX header concepts 275–8
- Object Push Profile (OPP) 308

- objects, active objects 6–7, 55–62, 102–3, 131–2, 208–9, 233–4, 268, 305, 311–12, 316–17
- OMA Device Management system 4, 8, 9, 14, 21, 22–3, 365–87, 392
 - adapters 367–87
 - commands 384–7
 - concepts 22–3, 365–87, 392
 - DDF 368–70, 379–87
 - definition 365–6
 - DFProperties 373–87
 - error values 382–4
 - essentials 367–72
 - high-level overview 366–7
 - historical background 365–6
 - management objects 368–87
 - management trees 368–87
 - specification 366–72
- OMA SyncML 21, 365–87
- On-The-Go (OTG) 64
- ‘one-box’ era 11
- Online Certificate Status Protocol 343
- Open 29–30, 31–3, 46–50, 51–2, 53–4, 144–6, 175, 183–6, 360–1
- Open Mobile Alliance 413
- open systems, Symbian OS 155
- OpenAsyncL 221–2
- OpenFStringL 354–5
- opening, sockets 29–30, 31–3, 46–50, 51–2, 53–4, 235–40
- OpenL 345–7, 353–4
- OpenStringL 354–5
- OpenTransactionL 348–9
- Opera Software AB 10
- OPP (Object Push Profile) 308, 336–7
- options, sockets 41–6, 89–91, 149–53
- OTA (over-the-air) 366
- OTG (On-The-Go) 64
- outgoing calls, telephony 207–9, 212–14
- over-the-air (OTA) 366
- overview 3–12, 13–23
- P&S (Publish & Subscribe) 89–91, 153
- packet ETEL 206
- packet loss performance measure 61, 198–203
- packet-switched wide-area communications links
 - see also* EDGE; GPRS; WCDMA
 - concepts 28, 30–1, 155–65, 205–6
- packets 28, 30–1, 155–65, 205–6, 274–81, 289–342
 - MTUs 94–5, 262, 308–16, 400–1
 - OBEX 274–81, 289–342
- pairability concept, Bluetooth 67–8, 88–9, 278
- PAN profile, Bluetooth 11–12, 16–17, 65, 83, 90, 155, 164–5, 167, 392–5, 408–10
- Parallax pods 395–6
- park mode, physical links 71–2, 92–3
- partner-only APIs 5–6
- passive connections 33, 36–7, 316–20
- passkeys, Bluetooth 68, 97–8
- passwords 68, 97–8, 157, 167–9, 278, 282–7, 333–42, 359–61, 373, 375
 - HTTP 359–61
 - IP 157, 167–9, 359–61
 - OBEX 278, 282–7, 333–42
- PC cards 393
- PCs 11, 320, 391–406
- PDAs 126
- PDP contexts, GPRS 157–8, 197–8, 203
- PDUs (protocol data units) 19, 74
- peers concepts, OBEX 274–5, 281, 284, 333–42
- per-channel flow control
 - IrDA 128–9
 - L2CAP 74
- per-technology bearer tables, IP 167–9
- performance issues
 - communications 61–2, 71, 197–203
- HTTP 356–61
- IP 164–5, 197–203
- messaging 233–5
- network bearer technologies 164–5
- OBEX 309–10, 329
- trade-off factors 61, 71
- types 61, 197–8
- persistent connections, HTTP 357–61
- phone IDs 208–11
 - see also* IMEIs
- physical channels
 - see also* piconet topology
 - Bluetooth 66, 70–2, 91–3
- physical links, Bluetooth 66, 68–72, 91–3, 112–18
- physical structure, SDBs 75–6
- piconet topology
 - see also* physical channels
 - Bluetooth 66, 69–72
- PIM applications 10
- PINs *see* passkeys
- pipelining 22, 347, 357–61
- PIPS sockets 163, 177–8
- Platform Security 6–7, 8–9, 12, 100, 128, 268–9, 338–42, 361
 - see also* security issues
- PLP 11
- plug-ins
 - see also* message-type modules
 - concepts 8, 15–21, 130, 232, 255–6, 361, 367, 379–81
- Point-to-Point Protocol (PPP) 18
- Pop 297–8, 311–13
- POP3 10, 20, 218, 220, 223–4, 246, 392
- PopAndDestroy 115, 170, 247, 250, 255, 355
- Portmon, SysInternals 404
- PortName 402–4, 406
- ports
 - see also* serial...
 - Bluetooth 80–2, 391–4, 397, 404
 - L2CAP 80–2
 - RFCOMM 80, 81–2
- Post command, HTTP 343–4, 349

- power management 10–11, 41, 63–6, 70–2, 91–3, 408–9
- PPP (Point-to-Point Protocol) 18
- PreventLowPowerModes 93
- PreventRoleSwitch 92–3
- primary contexts, GPRS 158, 198–9
- private directories, file attachments 253, 254–5
- processes, thread sharing 44–5
- profiles, Bluetooth 11–12, 16–17, 64–5, 81, 118–24
- programs
 - see also* applications
 - examples 52–62
- Progress 47–8
- progress notifications 47–8, 193–6
- ProgressNotification 47–8, 193–6
- PropertySet 348–9
- Protocol 187, 192–3
- protocol data units (PDUs) 19, 74
- protocol service multiplexers (PSMs), L2CAP ports 80–1, 116–18
- ProtocolDescriptorList 82, 105–6, 115–18
- protocols
 - Bluetooth 11–12, 16–17, 64–6, 73–4, 118–24
 - HTTP 4, 7–9, 11, 14, 21–3, 39, 50, 52–62, 128, 181, 241, 273–4, 277, 280, 292, 341, 343–64
 - IP 8, 11, 14, 18–19, 28, 30–3, 47, 49–50, 155–204, 205–6, 347, 355, 360–1, 396–404, 406, 409–10
 - IrDA 31–3, 130, 144–8, 273–4, 306–7
 - OBEX 273–342
 - types 31, 32–3, 80–2, 273–4
- ProtServ 241–2, 268–9
- proxy support, HTTP 355
- PRT files 16
- Psion
 - background 10–12
 - historical background 10–12
 - Series 5 organizer 10, 255, 289
- PSMs (protocol service multiplexers), L2CAP ports 80–1, 116–18
- PSTN 64
- Publish & Subscribe (P&S) 89–91, 153
- pure virtual functions, concepts 299–300, 334
- push messages 223–4, 241–2
- PushL 115, 168–70, 238, 247, 249, 295–9, 311–12
- Put 311–13, 315
- Put command, OBEX 274–5, 279–80, 284, 289, 292–3, 311–16, 322–6, 333, 339–41
- PutCompleteIndication 324–6, 339–41
- PUTing concepts 274
- PutPacketIndication 323–6, 329–30, 339–40
- PutRequestIndication 322–6, 339–41
- qikutils.lib 243
- QoS (quality of service) 12, 49, 156, 158, 197–203, 408
 - characteristics 197–8
 - concepts 197–203, 408
 - traffic classes 197
- QPhoneAppExternalInterface.h 214
- quality of service (QoS) 12, 49, 156, 158, 197–203, 408
 - characteristics 197–8
 - concepts 197–203, 408
 - traffic classes 197
- Query 134–40
- QueryCapability 259–71
- R (resource) classes 6, 49, 102
- R380 11
- RAM 10, 11, 227–8, 235, 289–91, 294, 300–4, 309–10
- range statistics, IrDA 126
- raw sockets
 - see also* sockets
 - concepts 31
- RBTPhysicalLinkAdapter 92–3, 102–3
- RBTRegistry 110–11
- RBTRegServ 110–11
- RBuf 40
- RComm 5, 17, 96, 129–30
- RConnection
 - see also* connection
 - management
 - concepts 8–9, 46–50, 51–2, 159–60, 162–204, 205–6, 345–9, 360–1
 - HTTP 345–9, 360–1
 - IP 159–60, 162–204
 - oddity 49
 - subconnections 49–50, 159–60, 198–203
 - telephony 205–6
- Read 41, 55, 181–3
- ReadDeviceData 211–12, 241–2, 268–9
- reading guide, chapters 9
- ReadStoreL 226, 230–1
- ReadUserData 211–12, 222, 268–9
- real-time control protocol (RTP/RTCP) 5–6, 65, 120, 367
- real-time systems 5–6, 61
- ReAllocL 40
- receiving data 37, 39–42, 53–62, 144–9, 175–93, 217–40, 246, 350–3
- IP 175–93
- sockets 37, 39–42, 53–62, 144–9, 175–93, 217–40, 246
- TCP 180–3
- receiving messages
 - see also* messaging
 - application-specific SMSs 235–40
 - concepts 217–40, 246
- recommended reading 6
- Record IDs 168–70
- Recv 30, 39–42, 45, 60–2, 102–3, 148–9, 181–3, 185–93
- RecvFrom 39–42, 185–6

- RecvOneOrMore 39–42, 55–6, 60–2, 181–3, 186–93
- RegisterServiceL 141
- RegisterThisService 147–8
- registration files, MTMs 269–71
- registry, Bluetooth 109–11
- ReleaseData 349, 351–3
- RemCon 17, 121
- remote control, Bluetooth 83, 118–24
- remote management
 - configuration settings 22–3
 - OMA Device Management system 4, 8, 9, 14, 21, 22–3, 365–87
- RemoteInfo 337
- Remove 143–4, 203
- RemoveField... 350
- RenegotiateHandshake 187–93
- repeater stations 112–18
- Replace command, OMA Device Management system 384–5
- Request 348–9
- request packets, OBEX client
 - applications 274–8, 279–87, 305–16, 320–42
- RequestCompleteIndication-Callback 340–1
- RequestIndicationCallback 340–1
- RequestIndicationCallback-WithError 340–1
- RequestSessionHeadersL 345–7
- Reset 299, 312–13, 349
- ResizeL 302–4
- RESOURCE 270–1
- resource files 269–71
- ResourceName 376
- response codes
 - HTTP 39, 181, 280
 - OBEX response packets 280, 282–7, 315–17, 323–5, 328, 330–2, 341
- response packets, OBEX server 274–8, 279–87, 316–41
- response phase, OBEX Get command 285–6, 326–30
- ResponseSessionHeadersL 345–7
- RF channels, piconet topology 70–2
- RFC1034 177
- RFC1918 159
- RFC2616 343, 348, 362
- RFC2617 344
- RFC2818 344
- RFCOMM 16–17, 30, 33, 74–6, 78–83, 96, 102–3, 105–6, 116–18, 274, 288, 308, 309, 318, 343, 348, 362
- Bluetooth 74–6, 78–83, 96, 102–3, 105–6, 116–18, 274, 288, 308, 309, 318
- concepts 78–83, 96, 102–3, 105–6, 116–18, 274, 288, 318
- critique 82
- flow control mechanisms 79–80, 309
- L2CAP uses 81–2, 96, 116–18
- ports 80, 81–2
- RFile 249–50, 253
- RGenericAgent 162
- RHostResolver
 - concepts 50–2, 84–91, 113–18, 131–2, 162–204
 - IP 162–204
- RHTTPConnection 345–7
- RHTTPConnectionInfo.h 347
- RHTTPFilterCollection 345–7, 363
- RHTTPHeaders 345–50
- RHTTPPropertySet 346–7
- RHTTPRequest 348–9
- RHTTPResponse 350
- RHTTPSession 345–9, 355, 363–4
- RHTTPTransaction 345–9, 353
- RHTTPTransactionProperty-Set 348–9
- RI 129
- rich text 252–4, 265–6
- ringtones 213, 217
- RNetDatabase
 - see also CSdpAgent
 - concepts 50–2, 132–4, 137–44
- RNif 162
- RNotifier 86–7
- roaming users 207, 410–11
- RollbackAtomicL 385–6
- ROM 10
- root server, concepts 15
- RPacketQoS 201–3
- RPointerArray 168–70, 292
- RReadStream 383–4
- RS-232 serial ports 19, 63, 79, 82, 96, 129–30, 394–7
 - see also serial server
- RSdp 103–9, 112–18
- RSdpDatabase 104–9, 112–18
- RSendAs 244–71
 - see also SendAs
- RSendAsMessage 244–71
- RServiceResolver 51–2
- RSocket 29–62, 80, 93–103, 130–53, 162–204, 205–6, 236–7
 - see also sockets
 - concepts 29–47, 80, 93–103, 130–1, 144–8, 162–204, 205–6, 236–7
 - critique 97
 - examples 52–62, 236–7
 - IP 162–204
 - process/thread sharing 44–5
 - telephony 205–6
- RSocketServ 29–62, 85–91, 114–18, 132, 137–41, 147–8, 165–75, 236–7, 360–1
 - see also sockets
 - concepts 29–30, 51–2, 132, 147–8, 236–7
 - examples 52–62, 236–7
- RString 353–5
- RStringF 353–6
- RStringPool 345–6, 353–5
- RStringTokenF 353–5
- RSubconnection, concepts 12, 49–50, 158, 198–203
- RSubConParameterBundle 199–203
- RTP/RTCP (real-time control protocol) 5–6, 65, 120, 367
- RTS 79, 130, 404

- RunDlgLD 255
 - RunError 58
 - RunL 53–61, 87–8, 110–11, 130–1, 137–41, 196, 208–9, 230, 233–4, 237–40, 305, 315
 - RWriteStream 385
 - S60
 - background 3–4, 5–6, 8–10, 12, 86–8, 98–100, 158–9, 163, 174, 213–14, 242–71, 392–4, 395–403
 - Bluetooth 84, 86–8, 98–100, 243, 392–4
 - emulator 392–4, 395–403
 - GPRS 158–9
 - IAPs 163
 - IP 158–9, 163, 164, 166, 174, 396–7
 - messaging 242–71
 - MTMs 257–60, 268–9
 - SDK 3–4, 392–4, 395–403
 - Send dialogs 243–4, 251–3, 258–60, 268–9
 - telephony 213–14, 403
 - sample code 8
 - SAR (segmentation-and-reassembly) 73–4, 93–4
 - saved messages folders 223
 - scatternet 70, 118
 - SCO/eSCO (synchronous connection oriented transports) 65, 66, 72–3
 - SCTS 387
 - SDAP (service discovery application profile) 109, 112–18
 - SDDBs (service discovery databases) 75–8, 104–9, 318–19
 - SDKConfig 397
 - SDKs (software development kits) 3–4, 211, 258, 392–403, 406
 - SDP (service discovery protocol) 65, 68, 74–8, 80–2, 83, 86, 90, 103–9, 112–18, 278, 287–8, 307–8, 318–19, 336–7
 - SDUs 74, 94–5
 - seamless roaming 410–11
 - secondary contexts, GPRS 156, 158, 198, 203
 - Secure ID 45
 - Secure Realtime Transport Protocol (SRTP) 6
 - secure sockets 12, 22, 186–93, 345, 348–9
 - security issues 6–7, 8–9, 12, 22, 71–2, 97–103, 157, 167–9, 186–93, 275, 277–8, 282–7, 333–42
 - see also* authentication; authorization; encryption; Platform Security
 - aspects 72
 - Bluetooth 71–2, 97–103, 278, 319–20
 - concepts 12, 22, 71–2, 97–103, 186–93, 248–50, 277–8, 282–4, 319–20, 333–42
 - data caging 248–50
 - dialogs 98–100
 - examples 101–2
 - HTTP 344, 348–9, 359–62
 - IP 12, 22, 186–93
 - IrDA 125, 128
 - messaging 210, 222, 268–9
 - modes 97–100
 - MTMs 268–9
 - OBEX 275, 277–8, 282–4, 289, 310–12, 319–20, 333–42
 - passwords 68, 97–8, 157, 167–9, 278, 282–7, 333–42, 359–61, 373, 375
 - settings examples 101–2
 - telephony 209–11
- segmentation-and-reassembly (SAR) 73–4, 93–4
- Send 37–9, 45, 59–60, 191–3
- Send dialogs
 - see also* CEikonEnv
- classes 250–5
- concepts 243–4, 250–71
- S60 243–4, 251–3, 258–9, 268–9
- SendAs 244, 255
- UIQ 243–4, 253–5, 258–9, 268–9
 - uses 244, 250–1
- SendAnSMSL 247–8
- SendAs 8, 20–1, 218, 220, 241–71, 305, 392, 419–20
 - see also* messaging
- capabilities 246–7, 268–9
- concepts 20–1, 220, 241–71, 305, 392, 419–20
- definition 242–3
- examples 247–8
- overview 244–5
- Send dialogs 244, 255
- technical description 246–50
- uses 244, 305
- SendAttachmentL 249–50
- sending data 20–1, 37–9, 53–62, 144–9, 175–93, 220, 241–71, 305, 392
- IP 175–6, 180–93
- sockets 37–9, 53–62, 144–9, 175–93
- TCP 180–3, 265
- sending messages
 - see also* messaging
 - concepts 241–71
- SendMessageAndCloseL 247–8
- SendMessageConfirmed 248
- SendRecv 45
- SendTo 37–9, 102–3, 184–6
- SendToFlickrL 266–8
- SendUI 242–3, 246, 251–3, 257–8
 - see also* Send dialogs
- sendui.lib 243
- SendWorkBench 253–5, 258, 416–17, 419–20
- sequential packet sockets
 - see also* sockets
 - concepts 30–3, 40, 73, 93–4, 145–9
- serial communications 5, 7, 10, 14–17, 19, 30, 33, 74–6, 78–83, 96, 102–3, 105–6, 116–18, 129, 274, 288, 308, 309, 318, 391–6, 403–6

- serial ports 5, 7, 10, 14, 19, 129, 391–6, 403–6
 - concept 5, 7, 19, 391–6, 403–6
 - emulator 391–6, 404–6
- serial server, concepts 14–15, 19
- server APIs, OBEX 290, 315–42
- server channels (RFCOMM ports), concepts 81
- server MTMs 20–1, 220–3, 246–50, 256–71
 - see also* message-type modules
 - client MTMs 264–5
 - concepts 246–50, 256–71
- server sockets, IrDA 144, 146–8
- ServerCert 190–3, 348
- servers 7–8, 14–19, 217, 220–40, 391
 - see also* serial. . . ; sockets
 - message server 217, 220–40
 - testing 391
- service class, Bluetooth 68–9, 74–8, 89–91, 104–9, 112–18
- service discovery
 - Bluetooth 68–9, 74–8, 80–2, 83, 86, 90, 103–9, 112–18, 287–8, 305–8, 318–22, 336, 409–10
 - IrDA 125–9, 133–40, 145–6, 288, 305–8, 317–22, 336
 - OBEX 287–8, 305–20
- service discovery application
 - profile (SDAP) 109, 112–18
- service discovery databases (SDDBs) 75–8, 104–9, 318–19
- service discovery protocol (SDP) 65, 68, 74–8, 80–2, 83, 86, 90, 103–9, 112–18, 278, 287–8, 307–8, 318–19, 336–7
- service entries, message store
 - structure 223–4
- ServiceEditL 263
- services, future prospects 407, 410–11
- ServiceSearch 113–18
- session initiation protocol (SIP) 6, 64
- Set 294–5
- SetActive 54–5, 87–8, 124, 131, 141–3, 145–6, 172, 184–6, 196, 311–15
- SetAddress 102, 177, 184–6
- SetAttributeName 140–3
- SetAttributePredictor-ListL 106–7
- SetAuthentication 100–2, 319
- SetAuthorisation 100–2, 319
- SetAvailableCipherSuites 186–93
- SetBearerSet 171
- SetBody 348–9
- SetBTAddr 84–91, 117
- SetByte 296
- SetByteSeqL 295–8, 311–16
- SetCallBack 334–5
- SetChallengeL 334–5
- SetClassName 140–3
- SetClientHandshake 186–93
- SetCommandTimeOut 341
- SetConfig 130
- SetConnectionCallBack 57–8
- SetCookie 356, 364
- SetDataBufL 302–4
- SetDenied 100–2
- SetDeviceSecurity 102
- SetDialogMode 186, 192
- SetDialogPreference 172
- SetDownLink. . . 200–3
- SetEncryption 100–2, 319
- SetEntryL 228–32
- SetFieldL 350, 355
- SetFourByte 296–8
- SetGenericSetL 201–3
- SetHeaderL 346, 354–5
- SetHintBitL 142–3
- SetIAC 84–91
- SetIapId 172
- SetLocalName 132–3
- SetLocalPort 37–8
- SetMask 300
- SetMax. . . 94–5
- SetNameL 292–3
- SetOpt 41–2, 45–6, 89–91, 149–51, 184–6, 187–93, 401–2
- SetParamL 350
- SetPasskeyMinLength 100–2
- SetPath 313–14
- SetPath command, OBEX 274–5, 279–80, 286, 289, 313–14, 330–2, 339–41
- SetPathIndication 330–2, 339–41
- SetPort 117, 183–6
- SetProperty 346–7
- SetProtocol 187, 192–3
- SetPutFinalResponse-Headers 325–6
- SetRawFieldL 350
- SetReceiveMtu 308–16
- SetRecordFilterL 106–7
- SetServerCert 186–93
- SetServerHandshake 186–93
- SetSocketCallback 56–7
- SetStatusL 383
- SetTargetChecking 337–8
- SetToCharString 140–3
- SetToInteger 140–3
- SetToOctetSeq 140–3
- SetTransmitMtu 308–16
- SetUid 100–2
- SetUnicodeL 297, 313
- SGSN (serving GPRS support node) 156–9
- ShareAuto 45
- ShareProtected 255
- short-range communications
 - technologies
 - see also* Bluetooth; IrDA; WiFi
 - concepts 28, 63–82
- ShowQueryAndSendL 253
- Shutdown 32, 42–6, 62, 182
- signalling stack 13–14, 19, 206
- signed software, MTMs 269
- SIM cards 157, 206, 208, 209–11
- simple1.cpp 52
- SIP (session initiation protocol) 6, 64
- Skype 161
- slaves, master devices 69–72, 112–18
- sliding-window flow control, TCP/IP 309

- smartphones
 - device class 88–9
 - future prospects 407–11
- SMS (short message service) 7–8, 14–15, 19, 20–1, 210, 217–40, 245–6, 247–8, 392, 403–4
 - see also* messaging
 - concepts 19, 20, 210, 217–40, 245–6, 247–8, 403–4
 - costs 243
 - intercepts 218, 235–40
 - locked applications 210
 - message store structure 223–6
 - receiving application-specific SMSs 235–40
 - SendAs 242–71
 - socket API 235–40
 - summary screens 218–21
 - video ringtones 217
 - weather reports 218–21, 235–40
- SMTP 10, 20, 50, 220, 245–6, 392
- sniff mode, physical links 71, 92, 112–18
- Socket 62
- sockets 7–8, 14–19, 27–62, 83, 93–4, 96–103, 114–18, 130–53, 155, 165–75, 180, 235–40, 345–7, 401–2
 - see also* Bluetooth; ESOCK; HTTP; IrDA; servers; SMS; TCP/IP
 - active connections 32, 33–6, 56–62, 196
 - addresses 31, 33–4, 50–2, 54–5, 86, 97, 127–31, 133–40
 - Bluetooth 83, 93–4, 96–103, 114–18
 - cancellations 32, 36, 37–9, 42–5, 185–6
 - closing 36, 37, 42–5, 62, 174–5, 182, 191–3, 194–6
 - concepts 7–8, 14–19, 27–62, 130–53, 155, 165–75, 180, 235–40, 345–7, 401–2
 - connecting 33–8, 53–62, 84–91, 104–6, 165–75, 180–3, 193–6, 237–40, 345–7
 - connection management 8–9, 27, 28, 46–50
 - descriptors 40
 - disconnecting 42–5, 62, 97, 111, 152–3, 173–5, 182, 194–6
 - examples 52–62, 117–18
 - historical background 28
 - L2CAP 93–5, 102–3
 - lifecycles 31–46
 - opening 29–30, 31–3, 46–50, 51–2, 53–4, 235–40
 - options 41–6, 89–91, 149–53
 - passive connections 33, 36–7
 - process/thread sharing 44–5
 - receiving data 37, 39–42, 53–62, 144–9, 175–93, 217–40, 246
 - secure sockets 12, 22, 186–93, 348–9
 - sending data 37–9, 53–62, 144–9, 175–93, 241–71
 - types 30–1, 41, 55–6, 145–6
- software development kits (SDKs) 3–4, 211, 258, 392–403, 406
- Sony-Ericsson 11, 409, 410
- specification phase, OBEX Get command 285–6, 326–30
- speed statistics, Bluetooth/IrDA contrasts 126
- SPP 83–124
- SQL statements 162–3
- SRTP (Secure Realtime Transport Protocol) 6
- SSIDs, WLAN 162–3, 167–9
- SSL/TLS 12, 186–93
- star topologies 69–70, 112
- Start... 46–50, 166–75, 186–93, 221–2, 233, 265, 318, 320–2, 330–2, 334, 385–6, StartAtomicL 385–6
- StartClientHandshake 186–93
- StartCommandL 265
- starting/stopping services, connection management 46–50, 165–75, 182, 185–6, 193–6
- StartProtocol 29–30
- StartRenegotiation 190–3
- state patterns, bus transport layer 114–18
- Stop 46–50, 173–5, 320–1
- StopListening 58
- StopProtocol 29–30
- stream sockets
 - see also* sockets
 - concepts 30–1, 40, 55–62, 96, 127–9, 148–9
- StreamCommittedL 385–6
- streaming media 73–4, 119–24
- StreamingSupport 385–6
- StringF 354–5
- StringL 354–5
- Stringpool, HTTP 344–5, 353–5
- subconnections
 - added sockets 203
 - concepts 49–50, 159–60, 198–203
- SubmitL 348–9
- Subscriber IDs 208–11
 - see also* IMSI numbers
- summary screens, messaging 217–35
- Supplementary Services, telephony 205–6
- Symbian Developer Network 413
- Symbian Ltd, formation 10–11
- Symbian OS 3–12, 13–23, 407–10
 - background 3–12, 13–23, 407–10
 - basics 6–7, 13–23
 - bearer technologies 160–2, 163–75, 193–6, 244–5, 391–2
 - Bluetooth 82–124, 391–4
 - continuous development 14, 407–10
 - default/implicit connections 163, 246
 - DM framework 4, 8, 9, 14, 21, 22–3, 365–87
 - documentation website 413

- Symbian OS (*continued*)
 - emulator 7–9, 15, 52, 244–5, 391–406
 - EPOC 15, 28, 289, 398, 404–6
 - future prospects 407–11
 - historical background 10–21, 289
 - HTTP 343–64
 - internals 14–15
 - Internet addresses 31, 54–6, 176–7
 - IP 159, 160–204, 396–403
 - IrDA implementation 129–53, 394–6
 - Library 9–10, 40, 211, 372–3
 - messaging 217–40, 241–71, 392
 - OBEX 273, 275, 289–342
 - OMA Device Management 4, 8, 9, 14, 21, 22–3, 365–87
 - open systems 155
 - overview 13–23, 82–3
 - popularity 12
 - security issues 6–7, 8–9, 12, 71–2, 97–103, 186–93, 333–42
 - v6.1 5, 11
 - v7.0 5, 11, 28, 46, 162–3, 211, 301, 343, 394
 - v8.0 11–12, 92, 366, 394
 - v8.1 5–6, 11–12, 82, 93
 - v9.0 8, 12, 13, 223, 248
 - v9.1 5–6, 8–9, 12, 13, 29, 89, 120, 128, 132, 162, 164, 211–12, 273, 301, 305, 316, 338, 340, 366, 399
 - v9.2 5–6, 8–9, 12, 28, 46, 89, 273, 290, 305, 316, 318, 321, 322, 328, 338–41
 - v9.3 365
 - version information 8–9
 - websites 7, 9–10, 413–14
- Symbian Press 411, 414
- synchronous connection oriented
 - transports (SCO/eSCO) 65, 66, 72–3
- SyncML 11, 21, 23, 343, 365–87
- SysInternals, Portmon 404
- system 394
- T (data-type) classes 6
- TAddrRange 109
- Target headers, OBEX 278–9, 284, 289, 311, 321–2, 333, 335–41
- target selector plug-in (TSP) 17
- TBTDevAddr 84–91
- TBTDeviceClass 90–1
- TBTDeviceSelectionParams 87–8
- TBTNamelessDevice 110–11
- TBTSecuritySettings 97–103
- TBTServiceSecurity... 102, 113–18
- TBTSockAddr 85–8, 97
- TBuf... 53, 60, 134, 140
- TBusState 114–18
- TCommDbConnPref 170–2
- TCommDbDialogPref 170–1
- TCommDbMultiConnPref 172
- TConnPref 46–7
- TCP/IP 4, 7, 9, 14–15, 18–19, 27–62, 66, 73, 128, 158, 176–7, 180–3, 265, 274, 309, 343, 345, 358, 401–2, 409–10
- Bluetooth 409–10
- concepts 18–19, 30–1, 50–2, 158, 176–7, 180–3, 265, 274, 309, 343, 358, 401–2, 409–10
- connecting 180–3, 358
- examples 52–62
- HTTP binding 274, 343, 345, 358
- port numbers 176–7
- sending/receiving data 180–3, 265
- sliding-window flow control 309
- state machines 180–1
- TDes... 48, 133–40
- technologies
 - bearer technologies 160–2, 163–75, 193–6, 244–5, 391–2, 410
 - exclusions 5–6
 - future prospects 407–11
 - scope 5–6
- telephony 4, 9, 14, 19, 69, 205–14, 391–2, 403–6, 413
 - see also ETel
 - concepts 19, 205–14, 391, 403–6
 - data access 205–6, 211–12
 - definition 205–6
 - emergency calls 213
 - emulator 391, 403–6
 - example application 208–9
 - high-level design 208–9
 - incoming calls 207–9, 213–14
 - locked applications 207, 209–11
 - notifications 213
 - outgoing calls 207–9, 212–14
 - overview 206
 - restrictions 211–14
 - roles 205–7
 - S60 213–14, 403
 - security issues 209–11
 - UIQ 213–14, 403
 - video 213–14
 - voice access 205–9, 211–14
- Telephony SYSTEM module (TSY) 14, 19, 206–14, 397–8, 403–4
- TError 382–3
- testing 7, 258, 391
- text shells 9
- TFilterConfiguration-Iterator 363
- TFT (traffic flow template) 158
- third-party developers 3–4, 217–18
- This 298–9
- threads 12, 29–30, 44–5, 305
- THTTPEvent 352
- THHTTPFilterHandle 347–9
- TIASDatabaseEntry 133–44
- TIASDataType 134–40
- TIASQuery 133–40
- TIASResponse 133–40
- time division multiplexing 70
- timers
 - IP networks 172–3, 174–5
 - OBEX 341
- TInetAddr 31, 54–6, 176–7

- TIquirySockAddr 31, 52, 84–91, 113–18
- TinyTP 18, 31, 33, 127–32, 145–6, 148–51, 274, 288, 306–7, 310
- TIp6Addr 177
- TIrdaAddr 31
- TIrdaSockAddr 131, 133–40, 145–8, 306–8
- TIrdaStatusCodes 153
- TL2CAPConfig... 94–5
- TL2CAPSockAddr 97, 113–18
- TLS 12, 22, 186–93, 376–9
- TMessageSummary 229–30
- TMSvEntry 225–6, 229–31, 265–6
- TMSvId 262, 266
- TMSvSessionEvent 221–2
- TNameAndLengthMask 300
- TNameEntry 51–2, 131, 178–80
- TNameRecord 151
- TNifProgress 195–6
- TObexBluetoothProtocol-Info 308, 319
- TObexBufferingDetails 301–4
- TObexConnectInfo 321–2
- TObexFilenameBackedBuffer 301–4
- TObexHeaderMask 292
- TObexIrProtocolInfo 306–7
- TObexProtocolInfo 305–20
- TObexProtocolPolicy 308–20
- TObexPureFileBuffer 301–4
- TObexRFileBackedBuffer 301–4
- TPckgBuf 43, 48, 49, 94–5, 239–40
- TPhoneIdV1 208
- TProtocolDesc 34–6
- TPtr... 40, 135
- traffic classes, 3G QoS 197
- traffic flow template (TFT) 158
- Transfer 45–6
- TransferCommandL 264–5
- TransportDownIndication 322, 333, 339–40
- transports
 - Bluetooth classes 66, 91, 114–18
 - OBEX 287–8, 305–6, 316–22, 336
- TransportUpIndication 321–2, 339–40
- TRAP 328–9
- TRemConCoreApiButton-Action 122–4
- TRequestStatus 34–6, 38–9, 46–7, 110–11, 166, 186–93, 228–9, 248–50, 267–8, 305–6, 310–16, 341
- TRfcommSockAddr 97
- TSdpAttributeID 104–5
- TSendingCapabilities 252–3
- TSessionPref 29–30
- TSetPathInfo 313–14, 330–2
- TShutdown 42–5
- TSmlDmAccessTypes 380–7
- TSmlDmMappingInfo 380–7
- TSmlMappingInfo 384–5
- TSmsAddr 237–8
- TSockAddr 31, 33–4, 54–5, 97, 176–7
- TSockXfrLength 41, 60, 184–6
- TSP (target selector plug-in) 17
- TState 145–8
- TSubConnectionInfo 49–50
- TSubscriberIdV1 208–9
- TSY (Telephony SYstem module) 14, 19, 206–14, 397–8, 403–4
- TsyForBearerAvailability 404
- TTargetChecking 337–8
- Type 107–9, 134–40, 295–6
- Type headers, OBEX 276–7
- UCS (Unicode) 11, 135–43, 276, 297–9, 313
- UDP 6, 18, 31, 33, 37, 66, 80–1, 120, 158, 161, 172, 176, 180–1, 183–6, 401
 - see also connectionless protocols
- UI see user interfaces
- UI MTMs 20–1, 256, 259, 262–3
- UID3s 269
- UIQ 3–10, 12, 84, 86–8, 98–100, 158–9, 163, 166, 174, 213–14, 242–71, 392–4, 395–403, 410, 414
 - background 3–10, 158–9, 163, 166, 174, 213–14, 242–71, 392–3, 395–403, 410, 414
- Bluetooth 84, 86–8, 98–100, 393
- Developer Community Portal 414
- emulator 392–3, 395–403
- GPRS 158–9
- IAPs 163
- IP 158–9, 163, 166, 174, 396–7
- messaging 242–71
- MTMs 257–60, 268–9
- SDK 3–4, 258, 392–3, 395–403
- Send dialogs 243–4, 253–5, 258–60, 268–9
- telephony 213–14, 403
- Ultra 282
- ultra-wideband radios (UWB) 408–9
- UMA 155
- UMTS signalling stack 13–14
- Unicode (UCS) 11, 135–43, 276, 297–9, 313
- universally unique identifiers (UUIDs) 75–8, 104–9, 113–18
- Unix 28
- UpdateAttributeL 104–5
- UpdateNextSummary 222
- URIs 22, 344, 368–87
- URLs 77
- USB 5–6, 11, 19, 21, 63, 64, 273–4, 289, 391, 394, 409
 - Bluetooth 391, 394, 409
 - device class 64
 - OBEX 21, 273–4, 289
 - OTG standard 64
- user interfaces (UI) 3–4, 9, 83–5, 208–9, 233–4, 241–71
 - see also S60; UIQ
 - functionality issues 4

- user interfaces (UI) (*continued*)
 - messaging 233–4, 241–71
 - types 4
- user-visible bandwidth,
 - performance issues 62
- User::LeaveIfError 37,
 - 53–4, 85, 90–1, 110–11, 131–2, 138–41, 145–8, 171–2, 178–9, 237, 247–50, 255, 297, 346
- UserName 375
- usernames, IP 157, 167–9, 359–61
- UserPasswordL 334–5
- User::WaitForRequest 249,
 - 290, 305
- UseTLS 376–9
- UTRAN 3G network 156
- UUIDs (universally unique identifiers) 75–8, 104–9, 113–18
- UWB (ultra-wideband radios) 408–9
- ValidHeaders 292
- vCard attachments 251, 273, 289,
 - 309
- VCF files 250
- VDP (video distribution profile) 119–24
- Vendor ID 45
- VerifyIMEICompleteL 209
- VerifyIMSICompleteL 209
- version information, Symbian OS 8–9
- video 17, 33, 63, 65, 83, 118–24,
 - 160–1, 163–5, 197–8, 213–14, 217
 - calls 213–14
 - SMS ringtones 217
 - telephony 213–14
- views, Bluetooth registry 110–11
- virtual functions, concepts 299–300
- virtual serial ports, concept 5, 7,
 - 19, 110, 403–4
- visible devices
 - Bluetooth 66–8
 - IrDA 132–3
- VisitAttributeValueL 116–18
- voice access, telephony 205–9,
 - 211–14
- Voice-overIP (VoIP) 49, 64–5
- WaitForAnyRequest 290
- WaitForRequest 249, 290, 305
- WaitForWeatherReport 237–40
- walled garden services, IP 155–61
- WCDMA 28, 161, 163–5, 175, 205
- weather reports 218–21, 235–40
- web browsers 10–11, 343–64
- web servers, examples 54–62
- web-cams 69
- websites OS 7, 9–10, 413–14
- while loop 298–9
- Who headers, OBEX 278–9, 284,
 - 289, 321–2, 333, 335–41
- WiFi (wireless LAN) 28, 161, 167
- Windows *see* Microsoft. . .
- WinPCap 397, 399–401
- winscw 393, 396, 398
- WinSock 397, 401–2
- WinTAP 397, 401
- WinTunnel 396–7
- Wireless Mobile Communications Device Class (WMCDC) 273–4
- WLAN 64–5, 91, 155, 159–60,
 - 162–5, 167–9, 175, 243, 359, 397, 399, 408–10
 - see also* Ethernet
 - costs 243
 - SSIDs 162–3, 167–9
- WMCDC (Wireless Mobile Communications Device Class) 273–4
- wrapper classes 27, 51, 52, 55–62,
 - 96–7, 102–3
- Write 38
- WriteComplete 59–60
- WriteDeviceData 241–2,
 - 268–9
- WriteUserData 222
- XML 77, 276, 286, 366–7, 370
- XMPP application 372–5,
 - 386–7
- XON/XOF 79, 129