DISTRIBUTED SYSTEMS

Concepts and Design

Fifth EditionThis page intentionally left blankDISTRIBUTED SYSTEMS

Concepts and Design

Fifth Edition

George Coulouris

Cambridge University

Jean Dollimore

formerly of Queen Mary,

University of London

Tim Kindberg

matter 2 media

Gordon Blair

Lancaster University

International Edition

contributions by:

Arup Kumar

Bhattacharjee

RCC Institute of Information

Technology, Kolkata

Soumen Mukherjee

RCC Institute of Information

Technology, KolkataEditorial Director: Marcia Horton

Editor-in-Chief: Michael Hirsch

Executive Editor: Matt Goldstein

Editorial Assistant: Chelsea Bell

Vice President, Marketing: Patrice Jones

Marketing Manager: Yezan Alayan

Marketing Coordinator: Kathryn Ferranti

Vice President, Production: Vince O’Brien

Managing Editor: Jeff Holcomb

Senior Production Project Manager: Marilyn Lloyd

Publisher, International Edition: Angshuman Chakraborty

Acquisitions Editor, International Edition: Arunabha Deb

Publishing Assistant, International Edition: Shokhi Shah

Senior Operations Supervisor: Alan Fischer

Manufacturing Buyer: Lisa McDowell

Art Director: Jayne Conte

Cover Designer: Suzanne Duda

Cover Image: Sky: © amygdala\_imagery; Kite: © Alamy; Mobile phone: © yasinguneysu/iStock

Media Editor: Daniel Sandin

Media Project Manager: Wanda Rockwell

Cover Printer: Lehigh-Phoenix Color

Pearson Education Limited

Edinburgh Gate

Harlow

Essex CM20 2JE

England

and Associated Companies throughout the world

Visit us on the World Wide Web at:

www.pearsoninternationaleditions.com

© Pearson Education Limited 2012

The rights of George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair to be identified as authors of this work have been

asserted by them in accordance with the Copyright, Designs and Patents Act 1988.

Authorized adaptation from the United States edition, entitled Distributed Systems, Concepts and Design, 5th edition, ISBN 978-0-

13-214301-1 by George Coulouris, Jean Dollimore, Tim Kindberg and Gordon Blair published by Pearson Education © 2012.

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 or otherwise, without either the prior written permission of the publisher or

a licence permitting restricted copying in the United Kingdom issued by the Copyright Licensing Agency Ltd, Saffron House, 6–10

Kirby Street, London EC1N 8TS.

All trademarks used herein are the property of their respective owners. The use of any trademark in this text does not vest in the

author or publisher any trademark ownership rights in such trademarks, nor does the use of such trademarks imply any affiliation

with or endorsement of this book by such owners.

Microsoft® and Windows® are registered trademarks of the Microsoft Corporation in the U.S.A. and other countries. Screen shots and icons

reprinted with permission from the Microsoft Corporation. This book is not sponsored or endorsed by or affiliated with the Microsoft Corporation.

British Library Cataloguing-in-Publication Data

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

10 9 8 7 6 5 4 3 2 1

14 13 12 11 10

Typeset in Times New Roman by authors using FrameMaker

Printed and bound by Edwards Brothers in The United States of America

The publisher's policy is to use paper manufactured from sustainable forests.

ISBN 10: 0-273-76059-9

ISBN 13: 978-0-273-76059-75

CONTENTS

PREFACE 11

1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS 17

1.1 Introduction 18

1.2 Examples of distributed systems 19

1.3 Trends in distributed systems 24

1.4 Focus on resource sharing 30

1.5 Challenges 32

1.6 Case study: The World Wide Web 42

1.7 Summary 49

2 SYSTEM MODELS 53

2.1 Introduction 54

2.2 Physical models 55

2.3 Architectural models 56

2.4 Fundamental models 77

2.5 Summary 92

3 NETWORKING AND INTERNETWORKING 97

3.1 Introduction 98

3.2 Types of network 102

3.3 Network principles 105

3.4 Internet protocols 122

3.5 Case studies: Ethernet, WiFi and Bluetooth 144

3.6 Summary 1576 CONTENTS

4 INTERPROCESS COMMUNICATION 161

4.1 Introduction 162

4.2 The API for the Internet protocols 163

4.3 External data representation and marshalling 174

4.4 Multicast communication 185

4.5 Network virtualization: Overlay networks 190

4.6 Case study: MPI 194

4.7 Summary 197

5 REMOTE INVOCATION 201

5.1 Introduction 202

5.2 Request-reply protocols 203

5.3 Remote procedure call 211

5.4 Remote method invocation 220

5.5 Case study: Java RMI 233

5.6 Summary 241

6 INDIRECT COMMUNICATION 245

6.1 Introduction 246

6.2 Group communication 248

6.3 Publish-subscribe systems 258

6.4 Message queues 270

6.5 Shared memory approaches 278

6.6 Summary 290

7 OPERATING SYSTEM SUPPORT 295

7.1 Introduction 296

7.2 The operating system layer 297

7.3 Protection 300

7.4 Processes and threads 302

7.5 Communication and invocation 319

7.6 Operating system architecture 330

7.7 Virtualization at the operating system level 334

7.8 Summary 347CONTENTS 7

8 DISTRIBUTED OBJECTS AND COMPONENTS 351

8.1 Introduction 352

8.2 Distributed objects 353

8.3 Case study: CORBA 356

8.4 From objects to components 374

8.5 Case studies: Enterprise JavaBeans and Fractal 380

8.6 Summary 394

9 WEB SERVICES 397

9.1 Introduction 398

9.2 Web services 400

9.3 Service descriptions and IDL for web services 416

9.4 A directory service for use with web services 420

9.5 XML security 422

9.6 Coordination of web services 427

9.7 Applications of web services 429

9.8 Summary 435

10 PEER-TO-PEER SYSTEMS 439

10.1 Introduction 440

10.2 Napster and its legacy 444

10.3 Peer-to-peer middleware 446

10.4 Routing overlays 449

10.5 Overlay case studies: Pastry, Tapestry 452

10.6 Application case studies: Squirrel, OceanStore, Ivy 465

10.7 Summary 474

11 SECURITY 479

11.1 Introduction 480

11.2 Overview of security techniques 488

11.3 Cryptographic algorithms 500

11.4 Digital signatures 509

11.5 Cryptography pragmatics 516

11.6 Case studies: Needham–Schroeder, Kerberos, TLS, 802.11 WiFi 519

11.7 Summary 5348 CONTENTS

12 DISTRIBUTED FILE SYSTEMS 537

12.1 Introduction 538

12.2 File service architecture 546

12.3 Case study: Sun Network File System 552

12.4 Case study: The Andrew File System 564

12.5 Enhancements and further developments 573

12.6 Summary 579

13 NAME SERVICES 581

13.1 Introduction 582

13.2 Name services and the Domain Name System 585

13.3 Directory services 600

13.4 Case study: The Global Name Service 601

13.5 Case study: The X.500 Directory Service 604

13.6 Summary 608

14 TIME AND GLOBAL STATES 611

14.1 Introduction 612

14.2 Clocks, events and process states 613

14.3 Synchronizing physical clocks 615

14.4 Logical time and logical clocks 623

14.5 Global states 626

14.6 Distributed debugging 635

14.7 Summary 642

15 COORDINATION AND AGREEMENT 645

15.1 Introduction 646

15.2 Distributed mutual exclusion 649

15.3 Elections 657

15.4 Coordination and agreement in group communication 662

15.5 Consensus and related problems 675

15.6 Summary 687CONTENTS 9

16 TRANSACTIONS AND CONCURRENCY CONTROL 691

16.1 Introduction 692

16.2 Transactions 695

16.3 Nested transactions 706

16.4 Locks 708

16.5 Optimistic concurrency control 723

16.6 Timestamp ordering 727

16.7 Comparison of methods for concurrency control 734

16.8 Summary 736

17 DISTRIBUTED TRANSACTIONS 743

17.1 Introduction 744

17.2 Flat and nested distributed transactions 744

17.3 Atomic commit protocols 747

17.4 Concurrency control in distributed transactions 756

17.5 Distributed deadlocks 759

17.6 Transaction recovery 767

17.7 Summary 777

18 REPLICATION 781

18.1 Introduction 782

18.2 System model and the role of group communication 784

18.3 Fault-tolerant services 791

18.4 Case studies of highly available services:

The gossip architecture, Bayou and Coda 798

18.5 Transactions with replicated data 818

18.6 Summary 830

19 MOBILE AND UBIQUITOUS COMPUTING 833

19.1 Introduction 834

19.2 Association 843

19.3 Interoperation 851

19.4 Sensing and context awareness 860

19.5 Security and privacy 873

19.6 Adaptation 882

19.7 Case study: Cooltown 887

19.8 Summary 89410 CONTENTS

20 DISTRIBUTED MULTIMEDIA SYSTEMS 897

20.1 Introduction 898

20.2 Characteristics of multimedia data 902

20.3 Quality of service management 903

20.4 Resource management 913

20.5 Stream adaptation 915

20.6 Case studies: Tiger, BitTorrent and End System Multicast 917

20.7 Summary 929

21 DESIGNING DISTRIBUTED SYSTEMS:

GOOGLE CASE STUDY 931

21.1 Introduction 932

21.2 Introducing the case study: Google 933

21.3 Overall architecture and design philosophy 938

21.4 Underlying communication paradigms 944

21.5 Data storage and coordination services 951

21.6 Distributed computation services 972

21.7 Summary 980

REFERENCES 983

INDEX 104111

PREFACE

This fifth edition of our textbook appears at a time when the Internet and the Web

continue to grow and have an impact on every aspect of our society. For example, the

introductory chapter of the book notes their impact on application areas as diverse as

finance and commerce, arts and entertainment and the emergence of the information

society more generally. It also highlights the very demanding requirements of

application domains such as web search and multiplayer online games. From a

distributed systems perspective, these developments are placing substantial new

demands on the underlying system infrastructure in terms of the range of applications

and the workloads and system sizes supported by many modern systems. Important

trends include the increasing diversity and ubiquity of networking technologies

(including the increasing importance of wireless networks), the inherent integration of

mobile and ubiquitous computing elements into the distributed systems infrastructure

New to the fifth edition

New chapters:

Indirect Communication: Covering group communication, publish-subscribe and

case studies on JavaSpaces, JMS, WebSphere and Message Queues.

Distributed Objects and Components: Covering component-based middleware and

case studies on Enterprise JavaBeans, Fractal and CORBA.

Designing Distributed Systems: Devoted to a major new case study on the Google

infrastructure.

Topics added to other chapters: Cloud computing, network virtualization, operating

system virtualization, message passing interface, unstructured peer-to-peer, tuple

spaces, loose coupling in relation to web services.

Other new case studies: Skype, Gnutella, TOTA, L2imbo, BitTorrent, End System

Multicast.

See the table on page 15 for further details of the changes.12 PREFACE

(leading to radically different physical architectures), the need to support multimedia

services and the emergence of the cloud computing paradigm, which challenges our

perspective of distributed systems services.

The book aims to provide an understanding of the principles on which the Internet

and other distributed systems are based; their architecture, algorithms and design; and

how they meet the demands of contemporary distributed applications. We begin with a

set of seven chapters that together cover the building blocks for a study of distributed

systems. The first two chapters provide a conceptual overview of the subject, outlining

the characteristics of distributed systems and the challenges that must be addressed in

their design: scalability, heterogeneity, security and failure handling being the most

significant. These chapters also develop abstract models for understanding process

interaction, failure and security. They are followed by other foundational chapters

devoted to the study of networking, interprocess communication, remote invocation,

indirect communication and operating system support.

The next set of chapters covers the important topic of middleware, examining

different approaches to supporting distributed applications including distributed objects

and components, web services and alternative peer-to-peer solutions. We then cover the

well-established topics of security, distributed file systems and distributed naming

before moving on to important data-related aspects including distributed transactions

and data replication. Algorithms associated with all these topics are covered as they arise

and also in separate chapters devoted to timing, coordination and agreement.

The book culminates in chapters that address the emerging areas of mobile and

ubiquitous computing and distributed multimedia systems before presenting a

substantial case study focusing on the design and implementation of the distributed

systems infrastructure that supports Google both in terms of core search functionality

and the increasing range of additional services offered by Google (for example, Gmail

and Google Earth). This last chapter has an important role in illustrating how all the

architectural concepts, algorithms and technologies introduced in the book can come

together in a coherent overall design for a given application domain.

Purposes and readership

The book is intended for use in undergraduate and introductory postgraduate courses. It

can equally be used for self-study. We take a top-down approach, addressing the issues

to be resolved in the design of distributed systems and describing successful approaches

in the form of abstract models, algorithms and detailed case studies of widely used

systems. We cover the field in sufficient depth and breadth to enable readers to go on to

study most research papers in the literature on distributed systems.

We aim to make the subject accessible to students who have a basic knowledge of

object-oriented programming, operating systems and elementary computer architecture.

The book includes coverage of those aspects of computer networks relevant to

distributed systems, including the underlying technologies for the Internet and for wide

area, local area and wireless networks. Algorithms and interfaces are presented

throughout the book in Java or, in a few cases, ANSI C. For brevity and clarity of

presentation, a form of pseudo-code derived from Java/C is also used.PREFACE 13

Organization of the book

The diagram shows the book’s chapters under seven main topic areas. It is intended to

provide a guide to the book’s structure and to indicate recommended navigation routes

for instructors wishing to provide, or readers wishing to achieve, understanding of the

various subfields of distributed system design.

References

The existence of the World Wide Web has changed the way in which a book such as this

can be linked to source material, including research papers, technical specifications and

standards. Many of the source documents are now available on the Web; some are

available only there. For reasons of brevity and readability, we employ a special form of

reference to web material that loosely resembles a URL: references such as

[www.omg.org] and [www.rsasecurity.com I] refer to documentation that is available

16 Transactions and Concurrency Control

17 Distributed Transactions

18 Replication

11 Security

12 Distributed File Systems

13 Name Services

System services

1 Characterization of

Distributed Systems

2 System Models

3 Networking and Internetworking

4 Interprocess Communication

5 Remote Invocation

6 Indirect Communication

7 Operating System Support

Foundations

14 Time and Global States

15 Coordination and Agreement

Distributed algorithms

Middleware

8 Dist. Objects and Components

9 Web Services

10 Peer-to-Peer Systems

19 Mobile and Ubiquitous Computing

20 Distributed Multimedia Systems

New challenges

Shared data

21 Designing Distributed Systems:

Google Case Study

Substantial case study14 PREFACE

only on the Web. They can be looked up in the reference list at the end of the book, but

the full URLs are given only in an online version of the reference list at the book’s web

site, www.cdk5.net/refs where they take the form of clickable links. Both versions of the

reference list include a more detailed explanation of this scheme.

Changes relative to the fourth edition

Before embarking on the writing of this new edition, we carried out a survey of teachers

who used the fourth edition. From the results, we identified the new material required

and a number of changes to be made. In addition, we recognized the increasing diversity

of distributed systems, particularly in terms of the range of architectural approaches

available to distributed systems developers today. This required significant changes to

the book, especially in the earlier (foundational) chapters.

Overall, this led to our writing three entirely new chapters, making substantial

changes to a number of other chapters and making numerous insertions throughout the

book to fold in new material. Many of the chapters have been changed to reflect new

information that has become available about the systems described. These changes are

summarized in the table below. To help teachers who have used the fourth edition,

wherever possible we have preserved the structure adopted from the previous edition.

Where material has been removed, we have placed this on our companion web site

together with material removed from previous editions. This includes the case studies

on ATM, interprocess communication in UNIX, CORBA (a shortened version of which

remains in Chapter 8), the Jini distributed events specification and Grid middleware

(featuring OGSA and the Globus toolkit), as well as the chapter on distributed shared

memory (a brief summary of which is now included in Chapter 6).

Some of the chapters in the book, such as the new chapter on indirect

communication (Chapter 6), cover a lot of material. Teachers may elect to cover the

broad spectrum before choosing two or three techniques to examine in more detail (for

example, group communication, given its foundational role, and publish-subscribe or

message queues, given their prevalence in commercial distributed systems).

The chapter ordering has been changed to accommodate the new material and to

reflect changes in the relative importance of certain topics. For a full understanding of

some topics readers may find it necessary to follow a forward reference. For example,

there is material in Chapter 9 on XML security techniques that will make better sense

once the sections that it references in Chapter 11 Security have been absorbed.

Acknowledgements

We are very grateful to the following teachers who participated in our survey: Guohong

Cao, Jose Fortes, Bahram Khalili, George Blank, Jinsong Ouyang, JoAnne Holliday,

George K. Thiruvathukal, Joel Wein, Tao Xie and Xiaobo Zhou.

We would like to thank the following people who reviewed the new chapters or

provided other substantial help: Rob Allen, Roberto Baldoni, John Bates, Tom Berson,

Lynne Blair, Geoff Coulson, Paul Grace, Andrew Herbert, David Hutchison, Laurent

Mathy, Rajiv Ramdhany, Richard Sharp, Jean-Bernard Stefani, Rip Sohan, FrancoisPREFACE 15

New chapters:

6 Indirect Communication Includes events and notification from 4th edition.

8 Distributed Objects and

Components

Includes a precised version of the CORBA case

study from the 4th edition.

21 Designing Distributed Systems Includes a major new case study on Google

Chapters which have undergone substantial changes:

1 Characterization of DS Significant restructuring of material

New Section 1.2: Examples of distributed systems

Section 1.3.4: Cloud computing introduced

2 System Models Significant restructuring of material

New Section 2.2: Physical models

Section 2.3: Major rewrite to reflect new book

content and associated architectural perspectives

4 Interprocess Communication Several updates

Client-server communication moved to Chapter 5

New Section 4.5: Network virtualization (includes

case study on Skype)

New Section 4.6: Case study on MPI

Case study on IPC in UNIX removed

5 Remote Invocation Significant restructuring of material

Client-server communication moved to here

Progression introduced from client-server

communication through RPC to RMI

Events and notification moved to Chapter 6

Chapters to which new material has been added/removed, but without structural changes:

3 Networking and Internetworking Several updates

Section 3.5: material on ATM removed

7 Operating System Support New Section 7.7: OS virtualization

9 Web Services Section 9.2: Discussion added on loose coupling

10 Peer-to-Peer Systems New Section 10.5.3: Unstructured peer-to-peer

(including a new case study on Gnutella)

15 Coordination and Agreement Material on group communication moved to Ch. 6

18 Replication Material on group communication moved to Ch. 6

19 Mobile and Ubiquitous Computing Section 19.3.1: New material on tuple spaces

(TOTA and L2imbo)

20 Distributed Multimedia Systems Section 20.6: New case studies added on

BitTorrent and End System Multicast

The remaining chapters have received only minor modifications.16 PREFACE

Taiani, Peter Triantafillou, Gareth Tyson and the late Sir Maurice Wilkes. We would

also like to thank the staff at Google who provided insights into the design rationale for

Google Infrastructure, namely: Mike Burrows, Tushar Chandra, Walfredo Cirne, Jeff

Dean, Sanjay Ghemawat, Andrea Kirmse and John Reumann.

Our copy editor, Rachel Head also provided outstanding support.

Web site

As before, we continue to maintain a web site with a wide range of material designed to

assist teachers and readers. This web site can be accessed via the URL:

www.cdk5.net

The web site includes:

Instructor’s Guide: We provide supporting material for teachers comprising:

• complete artwork of the book available as PowerPoint files;

• chapter-by-chapter teaching hints;

• solutions to the exercises, protected by a password available only to teachers.

Instructor resources for the International Edition are available at

www.pearsoninternationaleditions.com/coulouris

Reference list: The list of references that can be found at the end of the book is replicated

at the web site. The web version of the reference list includes active links for material

that is available online.

Errata list: A list of known errors in the book is maintained, with corrections. The errors

will be corrected when new impressions are printed and a separate errata list will be

provided for each impression. (Readers are encouraged to report any apparent errors

they encounter to the email address below.)

Supplementary material: We maintain a set of supplementary material for each chapter.

This consists of source code for the programs in the book and relevant reading material

that was present in previous editions of the book but was removed for reasons of space.

References to this supplementary material appear in the book with links such as

www.cdk5.net/ipc (the URL for supplementary material relating to the Interprocess

Communication chapter). Two entire chapters from the 4th edition are not present in this

one; they can be accessed at the URLs:

CORBA Case Study www.cdk5.net/corba

Distributed Shared Memory www.cdk5.net/dsm

George Coulouris

Jean Dollimore

Tim Kindberg

Gordon Blair

London, Bristol and Lancaster, 2011

authors@cdk5.net17

1

CHARACTERIZATION OF

DISTRIBUTED SYSTEMS

1.1 Introduction

1.2 Examples of distributed systems

1.3 Trends in distributed systems

1.4 Focus on resource sharing

1.5 Challenges

1.6 Case study: The World Wide Web

1.7 Summary

A distributed system is one in which components located at networked computers

communicate and coordinate their actions only by passing messages. This definition

leads to the following especially significant characteristics of distributed systems:

concurrency of components, lack of a global clock and independent failures of

components.

We look at several examples of modern distributed applications, including web

search, multiplayer online games and financial trading systems, and also examine the key

underlying trends driving distributed systems today: the pervasive nature of modern

networking, the emergence of mobile and ubiquitous computing, the increasing

importance of distributed multimedia systems, and the trend towards viewing distributed

systems as a utility. The chapter then highlights resource sharing as a main motivation for

constructing distributed systems. Resources may be managed by servers and accessed

by clients or they may be encapsulated as objects and accessed by other client objects.

The challenges arising from the construction of distributed systems are the

heterogeneity of their components, openness (which allows components to be added or

replaced), security, scalability – the ability to work well when the load or the number of

users increases – failure handling, concurrency of components, transparency and

providing quality of service. Finally, the Web is discussed as an example of a large-scale

distributed system and its main features are introduced.18 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

1.1 Introduction

Networks of computers are everywhere. The Internet is one, as are the many networks

of which it is composed. Mobile phone networks, corporate networks, factory networks,

campus networks, home networks, in-car networks – all of these, both separately and in

combination, share the essential characteristics that make them relevant subjects for

study under the heading distributed systems. In this book we aim to explain the

characteristics of networked computers that impact system designers and implementors

and to present the main concepts and techniques that have been developed to help in the

tasks of designing and implementing systems that are based on them.

We define a distributed system as one in which hardware or software components

located at networked computers communicate and coordinate their actions only by

passing messages. This simple definition covers the entire range of systems in which

networked computers can usefully be deployed.

Computers that are connected by a network may be spatially separated by any

distance. They may be on separate continents, in the same building or in the same room.

Our definition of distributed systems has the following significant consequences:

Concurrency: In a network of computers, concurrent program execution is the norm.

I can do my work on my computer while you do your work on yours, sharing

resources such as web pages or files when necessary. The capacity of the system to

handle shared resources can be increased by adding more resources (for example.

computers) to the network. We will describe ways in which this extra capacity can be

usefully deployed at many points in this book. The coordination of concurrently

executing programs that share resources is also an important and recurring topic.

No global clock: When programs need to cooperate they coordinate their actions by

exchanging messages. Close coordination often depends on a shared idea of the time

at which the programs’ actions occur. But it turns out that there are limits to the

accuracy with which the computers in a network can synchronize their clocks – there

is no single global notion of the correct time. This is a direct consequence of the fact

that the only communication is by sending messages through a network. Examples of

these timing problems and solutions to them will be described in Chapter 14.

Independent failures: All computer systems can fail, and it is the responsibility of

system designers to plan for the consequences of possible failures. Distributed systems

can fail in new ways. Faults in the network result in the isolation of the computers that

are connected to it, but that doesn’t mean that they stop running. In fact, the programs

on them may not be able to detect whether the network has failed or has become

unusually slow. Similarly, the failure of a computer, or the unexpected termination of

a program somewhere in the system (a crash), is not immediately made known to the

other components with which it communicates. Each component of the system can fail

independently, leaving the others still running. The consequences of this characteristic

of distributed systems will be a recurring theme throughout the book.

The prime motivation for constructing and using distributed systems stems from a desire

to share resources. The term ‘resource’ is a rather abstract one, but it best characterizes

the range of things that can usefully be shared in a networked computer system. ItSECTION 1.2 EXAMPLES OF DISTRIBUTED SYSTEMS 19

extends from hardware components such as disks and printers to software-defined

entities such as files, databases and data objects of all kinds. It includes the stream of

video frames that emerges from a digital video camera and the audio connection that a

mobile phone call represents.

The purpose of this chapter is to convey a clear view of the nature of distributed

systems and the challenges that must be addressed in order to ensure that they are

successful. Section 1.2 gives some illustrative examples of distributed systems, with

Section 1.3 covering the key underlying trends driving recent developments. Section 1.4

focuses on the design of resource-sharing systems, while Section 1.5 describes the key

challenges faced by the designers of distributed systems: heterogeneity, openness,

security, scalability, failure handling, concurrency, transparency and quality of service.

Section 1.6 presents a detailed case study of one very well known distributed system, the

World Wide Web, illustrating how its design supports resource sharing.

1.2 Examples of distributed systems

The goal of this section is to provide motivational examples of contemporary distributed

systems illustrating both the pervasive role of distributed systems and the great diversity

of the associated applications.

As mentioned in the introduction, networks are everywhere and underpin many

everyday services that we now take for granted: the Internet and the associated World

Wide Web, web search, online gaming, email, social networks, eCommerce, etc. To

illustrate this point further, consider Figure 1.1, which describes a selected range of key

commercial or social application sectors highlighting some of the associated established

or emerging uses of distributed systems technology.

As can be seen, distributed systems encompass many of the most significant

technological developments of recent years and hence an understanding of the

underlying technology is absolutely central to a knowledge of modern computing. The

figure also provides an initial insight into the wide range of applications in use today,

from relatively localized systems (as found, for example, in a car or aircraft) to globalscale systems involving millions of nodes, from data-centric services to processorintensive tasks, from systems built from very small and relatively primitive sensors to

those incorporating powerful computational elements, from embedded systems to ones

that support a sophisticated interactive user experience, and so on.

We now look at more specific examples of distributed systems to further illustrate

the diversity and indeed complexity of distributed systems provision today.

1.2.1 Web search

Web search has emerged as a major growth industry in the last decade, with recent

figures indicating that the global number of searches has risen to over 10 billion per

calendar month. The task of a web search engine is to index the entire contents of the

World Wide Web, encompassing a wide range of information styles including web

pages, multimedia sources and (scanned) books. This is a very complex task, as current

estimates state that the Web consists of over 63 billion pages and one trillion unique web20 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

Figure 1.1 Selected application domains and associated networked applications

Finance and commerce The growth of eCommerce as exemplified by companies such as

Amazon and eBay, and underlying payments technologies such as

PayPal; the associated emergence of online banking and trading and

also complex information dissemination systems for financial markets.

The information society The growth of the World Wide Web as a repository of information and

knowledge; the development of web search engines such as Google

and Yahoo to search this vast repository; the emergence of digital

libraries and the large-scale digitization of legacy information sources

such as books (for example, Google Books); the increasing

significance of user-generated content through sites such as YouTube,

Wikipedia and Flickr; the emergence of social networking through

services such as Facebook and MySpace.

Creative industries and

entertainment

The emergence of online gaming as a novel and highly interactive form

of entertainment; the availability of music and film in the home

through networked media centres and more widely in the Internet via

downloadable or streaming content; the role of user-generated content

(as mentioned above) as a new form of creativity, for example via

services such as YouTube; the creation of new forms of art and entertainment enabled by emergent (including networked) technologies.

Healthcare The growth of health informatics as a discipline with its emphasis on

online electronic patient records and related issues of privacy; the

increasing role of telemedicine in supporting remote diagnosis or more

advanced services such as remote surgery (including collaborative

working between healthcare teams); the increasing application of

networking and embedded systems technology in assisted living, for

example for monitoring the elderly in their own homes.

Education The emergence of e-learning through for example web-based tools

such as virtual learning environments; associated support for distance

learning; support for collaborative or community-based learning.

Transport and logistics The use of location technologies such as GPS in route finding systems

and more general traffic management systems; the modern car itself as

an example of a complex distributed system (also applies to other

forms of transport such as aircraft); the development of web-based map

services such as MapQuest, Google Maps and Google Earth.

Science The emergence of the Grid as a fundamental technology for eScience,

including the use of complex networks of computers to support the

storage, analysis and processing of (often very large quantities of)

scientific data; the associated use of the Grid as an enabling technology

for worldwide collaboration between groups of scientists.

Environmental management The use of (networked) sensor technology to both monitor and manage

the natural environment, for example to provide early warning of

natural disasters such as earthquakes, floods or tsunamis and to coordinate emergency response; the collation and analysis of global

environmental parameters to better understand complex natural

phenomena such as climate change.SECTION 1.2 EXAMPLES OF DISTRIBUTED SYSTEMS 21

addresses. Given that most search engines analyze the entire web content and then carry

out sophisticated processing on this enormous database, this task itself represents a

major challenge for distributed systems design.

Google, the market leader in web search technology, has put significant effort into

the design of a sophisticated distributed system infrastructure to support search (and

indeed other Google applications and services such as Google Earth). This represents

one of the largest and most complex distributed systems installations in the history of

computing and hence demands close examination. Highlights of this infrastructure

include:

• an underlying physical infrastructure consisting of very large numbers of

networked computers located at data centres all around the world;

• a distributed file system designed to support very large files and heavily optimized

for the style of usage required by search and other Google applications (especially

reading from files at high and sustained rates);

• an associated structured distributed storage system that offers fast access to very

large datasets;

• a lock service that offers distributed system functions such as distributed locking

and agreement;

• a programming model that supports the management of very large parallel and

distributed computations across the underlying physical infrastructure.

Further details on Google’s distributed systems services and underlying communications support can be found in Chapter 21, a compelling case study of a modern distributed system in action.

1.2.2 Massively multiplayer online games (MMOGs)

Massively multiplayer online games offer an immersive experience whereby very large

numbers of users interact through the Internet with a persistent virtual world. Leading

examples of such games include Sony’s EverQuest II and EVE Online from the Finnish

company CCP Games. Such worlds have increased significantly in sophistication and

now include, complex playing arenas (for example EVE, Online consists of a universe

with over 5,000 star systems) and multifarious social and economic systems. The

number of players is also rising, with systems able to support over 50,000 simultaneous

online players (and the total number of players perhaps ten times this figure).

The engineering of MMOGs represents a major challenge for distributed systems

technologies, particularly because of the need for fast response times to preserve the user

experience of the game. Other challenges include the real-time propagation of events to

the many players and maintaining a consistent view of the shared world. This therefore

provides an excellent example of the challenges facing modern distributed systems

designers.

A number of solutions have been proposed for the design of massively multiplayer

online games:

• Perhaps surprisingly, the largest online game, EVE Online, utilises a client-server

architecture where a single copy of the state of the world is maintained on a22 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

centralized server and accessed by client programs running on players’ consoles

or other devices. To support large numbers of clients, the server is a complex

entity in its own right consisting of a cluster architecture featuring hundreds of

computer nodes (this client-server approach is discussed in more detail in Section

1.4 and cluster approaches are discussed in Section 1.3.4). The centralized

architecture helps significantly in terms of the management of the virtual world

and the single copy also eases consistency concerns. The goal is then to ensure fast

response through optimizing network protocols and ensuring a rapid response to

incoming events. To support this, the load is partitioned by allocating individual

‘star systems’ to particular computers within the cluster, with highly loaded star

systems having their own dedicated computer and others sharing a computer.

Incoming events are directed to the right computers within the cluster by keeping

track of movement of players between star systems.

• Other MMOGs adopt more distributed architectures where the universe is

partitioned across a (potentially very large) number of servers that may also be

geographically distributed. Users are then dynamically allocated a particular

server based on current usage patterns and also the network delays to the server

(based on geographical proximity for example). This style of architecture, which

is adopted by EverQuest, is naturally extensible by adding new servers.

• Most commercial systems adopt one of the two models presented above, but

researchers are also now looking at more radical architectures that are not based

on client-server principles but rather adopt completely decentralized approaches

based on peer-to-peer technology where every participant contributes resources

(storage and processing) to accommodate the game. Further consideration of peerto-peer solutions is deferred until Chapters 2 and 10).

1.2.3 Financial trading

As a final example, we look at distributed systems support for financial trading markets.

The financial industry has long been at the cutting edge of distributed systems

technology with its need, in particular, for real-time access to a wide range of

information sources (for example, current share prices and trends, economic and

political developments). The industry employs automated monitoring and trading

applications (see below).

Note that the emphasis in such systems is on the communication and processing

of items of interest, known as events in distributed systems, with the need also to deliver

events reliably and in a timely manner to potentially very large numbers of clients who

have a stated interest in such information items. Examples of such events include a drop

in a share price, the release of the latest unemployment figures, and so on. This requires

a very different style of underlying architecture from the styles mentioned above (for

example client-server), and such systems typically employ what are known as

distributed event-based systems. We present an illustration of a typical use of such

systems below and return to this important topic in more depth in Chapter 6.

Figure 1.2 illustrates a typical financial trading system. This shows a series of

event feeds coming into a given financial institution. Such event feeds share theSECTION 1.2 EXAMPLES OF DISTRIBUTED SYSTEMS 23

following characteristics. Firstly, the sources are typically in a variety of formats, such

as Reuters market data events and FIX events (events following the specific format of

the Financial Information eXchange protocol), and indeed from different event

technologies, thus illustrating the problem of heterogeneity as encountered in most

distributed systems (see also Section 1.5.1). The figure shows the use of adapters which

translate heterogeneous formats into a common internal format. Secondly, the trading

system must deal with a variety of event streams, all arriving at rapid rates, and often

requiring real-time processing to detect patterns that indicate trading opportunities. This

used to be a manual process but competitive pressures have led to increasing automation

in terms of what is known as Complex Event Processing (CEP), which offers a way of

composing event occurrences together into logical, temporal or spatial patterns.

This approach is primarily used to develop customized algorithmic trading

strategies covering both buying and selling of stocks and shares, in particular looking

for patterns that indicate a trading opportunity and then automatically responding by

placing and managing orders. As an example, consider the following script:

WHEN

MSFT price moves outside 2% of MSFT Moving Average

FOLLOWED-BY (

MyBasket moves up by 0.5%

AND

HPQ’s price moves up by 5%

OR

MSFT’s price moves down by 2%

)

)

ALL WITHIN

any 2 minute time period

THEN

BUY MSFT

SELL HPQ

Figure 1.2 An example financial trading system

FIX

Gateway

Complex

Event Processing

Engine

FIX

Adapter

Reuters

Adapter

Reuters

Gateway

FIX events Reuters events

Trading strategies24 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

This script is based on the functionality provided by Apama [www.progress.com], a

commercial product in the financial world originally developed out of research carried

out at the University of Cambridge. The script detects a complex temporal sequence

based on the share prices of Microsoft, HP and a basket of other share prices, resulting

in decisions to buy or sell particular shares.

This style of technology is increasingly being used in other areas of financial

systems including the monitoring of trading activity to manage risk (in particular,

tracking exposure), to ensure compliance with regulations and to monitor for patterns of

activity that might indicate fraudulent transactions. In such systems, events are typically

intercepted and passed through what is equivalent to a compliance and risk firewall

before being processed (see also the discussion of firewalls in Section 1.3.1 below).

1.3 Trends in distributed systems

Distributed systems are undergoing a period of significant change and this can be traced

back to a number of influential trends:

• the emergence of pervasive networking technology;

• the emergence of ubiquitous computing coupled with the desire to support user

mobility in distributed systems;

• the increasing demand for multimedia services;

• the view of distributed systems as a utility.

1.3.1 Pervasive networking and the modern Internet

The modern Internet is a vast interconnected collection of computer networks of many

different types, with the range of types increasing all the time and now including, for

example, a wide range of wireless communication technologies such as WiFi, WiMAX,

Bluetooth (see Chapter 3) and third-generation mobile phone networks. The net result is

that networking has become a pervasive resource and devices can be connected (if

desired) at any time and in any place.

Figure 1.3 illustrates a typical portion of the Internet. Programs running on the

computers connected to it interact by passing messages, employing a common means of

communication. The design and construction of the Internet communication

mechanisms (the Internet protocols) is a major technical achievement, enabling a

program running anywhere to address messages to programs anywhere else and

abstracting over the myriad of technologies mentioned above.

The Internet is also a very large distributed system. It enables users, wherever they

are, to make use of services such as the World Wide Web, email and file transfer.

(Indeed, the Web is sometimes incorrectly equated with the Internet.) The set of services

is open-ended – it can be extended by the addition of server computers and new types of

service. The figure shows a collection of intranets – subnetworks operated by companies

and other organizations and typically protected by firewalls. The role of a firewall is to

protect an intranet by preventing unauthorized messages from leaving or entering. ASECTION 1.3 TRENDS IN DISTRIBUTED SYSTEMS 25

firewall is implemented by filtering incoming and outgoing messages. Filtering might

be done by source or destination, or a firewall might allow only those messages related

to email and web access to pass into or out of the intranet that it protects. Internet Service

Providers (ISPs) are companies that provide broadband links and other types of

connection to individual users and small organizations, enabling them to access services

anywhere in the Internet as well as providing local services such as email and web

hosting. The intranets are linked together by backbones. A backbone is a network link

with a high transmission capacity, employing satellite connections, fibre optic cables

and other high-bandwidth circuits.

Note that some organizations may not wish to connect their internal networks to

the Internet at all. For example, police and other security and law enforcement agencies

are likely to have at least some internal intranets that are isolated from the outside world

(the most effective firewall possible – the absence of any physical connections to the

Internet). Firewalls can also be problematic in distributed systems by impeding

legitimate access to services when resource sharing between internal and external users

is required. Hence, firewalls must often be complemented by more fine-grained

mechanisms and policies, as discussed in Chapter 11.

The implementation of the Internet and the services that it supports has entailed

the development of practical solutions to many distributed system issues (including

most of those defined in Section 1.5). We shall highlight those solutions throughout the

book, pointing out their scope and their limitations where appropriate.

Figure 1.3 A typical portion of the Internet

intranet

ISP

desktop computer:

backbone

backbone

satellite link

server:

backbone

network link:26 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

1.3.2 Mobile and ubiquitous computing

Technological advances in device miniaturization and wireless networking have led

increasingly to the integration of small and portable computing devices into distributed

systems. These devices include:

• Laptop computers.

• Handheld devices, including mobile phones, smart phones, GPS-enabled devices,

pagers, personal digital assistants (PDAs), video cameras and digital cameras.

• Wearable devices, such as smart watches with functionality similar to a PDA.

• Devices embedded in appliances such as washing machines, hi-fi systems, cars

and refrigerators.

The portability of many of these devices, together with their ability to connect

conveniently to networks in different places, makes mobile computing possible. Mobile

computing is the performance of computing tasks while the user is on the move, or

visiting places other than their usual environment. In mobile computing, users who are

away from their ‘home’ intranet (the intranet at work, or their residence) are still

provided with access to resources via the devices they carry with them. They can

continue to access the Internet; they can continue to access resources in their home

intranet; and there is increasing provision for users to utilize resources such as printers

or even sales points that are conveniently nearby as they move around. The latter is also

known as location-aware or context-aware computing. Mobility introduces a number of

challenges for distributed systems, including the need to deal with variable connectivity

and indeed disconnection, and the need to maintain operation in the face of device

mobility (see the discussion on mobility transparency in Section 1.5.7).

Ubiquitous computing is the harnessing of many small, cheap computational

devices that are present in users’ physical environments, including the home, office and

even natural settings. The term ‘ubiquitous’ is intended to suggest that small computing

devices will eventually become so pervasive in everyday objects that they are scarcely

noticed. That is, their computational behaviour will be transparently and intimately tied

up with their physical function.

The presence of computers everywhere only becomes useful when they can

communicate with one another. For example, it may be convenient for users to control

their washing machine or their entertainment system from their phone or a ‘universal

remote control’ device in the home. Equally, the washing machine could notify the user

via a smart badge or phone when the washing is done.

Ubiquitous and mobile computing overlap, since the mobile user can in principle

benefit from computers that are everywhere. But they are distinct, in general. Ubiquitous

computing could benefit users while they remain in a single environment such as the

home or a hospital. Similarly, mobile computing has advantages even if it involves only

conventional, discrete computers and devices such as laptops and printers.

Figure 1.4 shows a user who is visiting a host organization. The figure shows the

user’s home intranet and the host intranet at the site that the user is visiting. Both

intranets are connected to the rest of the Internet.

The user has access to three forms of wireless connection. Their laptop has a

means of connecting to the host’s wireless LAN. This network provides coverage of aSECTION 1.3 TRENDS IN DISTRIBUTED SYSTEMS 27

few hundred metres (a floor of a building, say). It connects to the rest of the host intranet

via a gateway or access point. The user also has a mobile (cellular) telephone, which is

connected to the Internet. The phone gives access to the Web and other Internet services,

constrained only by what can be presented on its small display, and may also provide

location information via built-in GPS functionality. Finally, the user carries a digital

camera, which can communicate over a personal area wireless network (with range up

to about 10m) with a device such as a printer.

With a suitable system infrastructure, the user can perform some simple tasks in

the host site using the devices they carry. While journeying to the host site, the user can

fetch the latest stock prices from a web server using the mobile phone and can also use

the built-in GPS and route finding software to get directions to the site location. During

the meeting with their hosts, the user can show them a recent photograph by sending it

from the digital camera directly to a suitably enabled (local) printer or projector in the

meeting room (discovered using a location service). This requires only the wireless link

between the camera and printer or projector. And they can in principle send a document

from their laptop to the same printer, utilizing the wireless LAN and wired Ethernet links

to the printer.

This scenario demonstrates the need to support spontaneous interoperation,

whereby associations between devices are routinely created and destroyed – for example

by locating and using the host’s devices, such as printers. The main challenge applying

to such situations is to make interoperation fast and convenient (that is, spontaneous)

even though the user is in an environment they may never have visited before. That

means enabling the visitor’s device to communicate on the host network, and

associating the device with suitable local services – a process called service discovery.

Mobile and ubiquitous computing represent lively areas of research, and the

various dimensions mentioned above are discussed in depth in Chapter 19.

Figure 1.4 Portable and handheld devices in a distributed system

Laptop

Mobile

Printer

Camera

Internet

Host intranet Home intranet

Wireless LAN

phone

Host site

3G phone network

GPS satellite signal28 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

1.3.3 Distributed multimedia systems

Another important trend is the requirement to support multimedia services in distributed

systems. Multimedia support can usefully be defined as the ability to support a range of

media types in an integrated manner. One can expect a distributed system to support the

storage, transmission and presentation of what are often referred to as discrete media

types, such as pictures or text messages. A distributed multimedia system should be able

to perform the same functions for continuous media types such as audio and video; that

is, it should be able to store and locate audio or video files, to transmit them across the

network (possibly in real time as the streams emerge from a video camera), to support

the presentation of the media types to the user and optionally also to share the media

types across a group of users.

The crucial characteristic of continuous media types is that they include a

temporal dimension, and indeed, the integrity of the media type is fundamentally

dependent on preserving real-time relationships between elements of a media type. For

example, in a video presentation it is necessary to preserve a given throughput in terms

of frames per second and, for real-time streams, a given maximum delay or latency for

the delivery of frames (this is one example of quality of service, discussed in more detail

in Section 1.5.8).

The benefits of distributed multimedia computing are considerable in that a wide

range of new (multimedia) services and applications can be provided on the desktop,

including access to live or pre-recorded television broadcasts, access to film libraries

offering video-on-demand services, access to music libraries, the provision of audio and

video conferencing facilities and integrated telephony features including IP telephony

or related technologies such as Skype, a peer-to-peer alternative to IP telephony (the

distributed system infrastructure underpinning Skype is discussed in Section 4.5.2).

Note that this technology is revolutionary in challenging manufacturers to rethink many

consumer devices. For example, what is the core home entertainment device of the

future – the computer, the television, or the games console?

Webcasting is an application of distributed multimedia technology. Webcasting is

the ability to broadcast continuous media, typically audio or video, over the Internet. It

is now commonplace for major sporting or music events to be broadcast in this way,

often attracting large numbers of viewers (for example, the Live8 concert in 2005

attracted around 170,000 simultaneous users at its peak).

Distributed multimedia applications such as webcasting place considerable

demands on the underlying distributed infrastructure in terms of:

• providing support for an (extensible) range of encoding and encryption formats,

such as the MPEG series of standards (including for example the popular MP3

standard otherwise known as MPEG-1, Audio Layer 3) and HDTV;

• providing a range of mechanisms to ensure that the desired quality of service can

be met;

• providing associated resource management strategies, including appropriate

scheduling policies to support the desired quality of service;

• providing adaptation strategies to deal with the inevitable situation in open

systems where quality of service cannot be met or sustained.

Further discussion of such mechanisms can be found in Chapter 20.SECTION 1.3 TRENDS IN DISTRIBUTED SYSTEMS 29

1.3.4 Distributed computing as a utility

With the increasing maturity of distributed systems infrastructure, a number of

companies are promoting the view of distributed resources as a commodity or utility,

drawing the analogy between distributed resources and other utilities such as water or

electricity. With this model, resources are provided by appropriate service suppliers and

effectively rented rather than owned by the end user. This model applies to both physical

resources and more logical services:

• Physical resources such as storage and processing can be made available to

networked computers, removing the need to own such resources on their own. At

one end of the spectrum, a user may opt for a remote storage facility for file

storage requirements (for example, for multimedia data such as photographs,

music or video) and/or for backups. Similarly, this approach would enable a user

to rent one or more computational nodes, either to meet their basic computing

needs or indeed to perform distributed computation. At the other end of the

spectrum, users can access sophisticated data centres (networked facilities

offering access to repositories of often large volumes of data to users or

organizations) or indeed computational infrastructure using the sort of services

now provided by companies such as Amazon and Google. Operating system

virtualization is a key enabling technology for this approach, implying that users

may actually be provided with services by a virtual rather than a physical node.

This offers greater flexibility to the service supplier in terms of resource

management (operating system virtualization is discussed in more detail in

Chapter 7).

• Software services (as defined in Section 1.4) can also be made available across the

global Internet using this approach. Indeed, many companies now offer a

comprehensive range of services for effective rental, including services such as

email and distributed calendars. Google, for example, bundles a range of business

services under the banner Google Apps [www.google.com I]. This development

is enabled by agreed standards for software services, for example as provided by

web services (see Chapter 9).

The term cloud computing is used to capture this vision of computing as a utility. A

cloud is defined as a set of Internet-based application, storage and computing services

sufficient to support most users’ needs, thus enabling them to largely or totally dispense

with local data storage and application software (see Figure 1.5). The term also

promotes a view of everything as a service, from physical or virtual infrastructure

through to software, often paid for on a per-usage basis rather than purchased. Note that

cloud computing reduces requirements on users’ devices, allowing very simple desktop

or portable devices to access a potentially wide range of resources and services.

Clouds are generally implemented on cluster computers to provide the necessary

scale and performance required by such services. A cluster computer is a set of

interconnected computers that cooperate closely to provide a single, integrated highperformance computing capability. Building on projects such as the NOW (Network of

Workstations) Project at Berkeley [Anderson et al. 1995, now.cs.berkeley.edu] and

Beowulf at NASA [www.beowulf.org], the trend is towards utilizing commodity

hardware both for the computers and for the interconnecting networks. Most clusters30 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

consist of commodity PCs running a standard (sometimes cut-down) version of an

operating system such as Linux, interconnected by a local area network. Companies

such as HP, Sun and IBM offer blade solutions. Blade servers are minimal

computational elements containing for example processing and (main memory) storage

capabilities. A blade system consists of a potentially large number of blade servers

contained within a blade enclosure. Other elements such as power, cooling, persistent

storage (disks), networking and displays, are provided either by the enclosure or through

virtualized solutions (discussed in Chapter 7). Through this solution, individual blade

servers can be much smaller and also cheaper to produce than commodity PCs.

The overall goal of cluster computers is to provide a range of cloud services,

including high-performance computing capabilities, mass storage (for example through

data centres), and richer application services such as web search (Google, for example

relies on a massive cluster computer architecture to implement its search engine and

other services, as discussed in Chapter 21).

Grid computing (as discussed in Chapter 9, Section 9.7.2) can also be viewed as

a form of cloud computing. The terms are largely synonymous and at times ill-defined,

but Grid computing can generally be viewed as a precursor to the more general paradigm

of cloud computing with a bias towards support for scientific applications.

1.4 Focus on resource sharing

Users are so accustomed to the benefits of resource sharing that they may easily

overlook their significance. We routinely share hardware resources such as printers, data

resources such as files, and resources with more specific functionality such as search

engines.

Figure 1.5 Cloud computing

Internet

Application services

Storage services

Computational services

ClientsSECTION 1.4 FOCUS ON RESOURCE SHARING 31

Looked at from the point of view of hardware provision, we share equipment such

as printers and disks to reduce costs. But of far greater significance to users is the sharing

of the higher-level resources that play a part in their applications and in their everyday

work and social activities. For example, users are concerned with sharing data in the

form of a shared database or a set of web pages – not the disks and processors on which

they are implemented. Similarly, users think in terms of shared resources such as a

search engine or a currency converter, without regard for the server or servers that

provide these.

In practice, patterns of resource sharing vary widely in their scope and in how

closely users work together. At one extreme, a search engine on the Web provides a

facility to users throughout the world, users who need never come into contact with one

another directly. At the other extreme, in computer-supported cooperative working

(CSCW), a group of users who cooperate directly share resources such as documents in

a small, closed group. The pattern of sharing and the geographic distribution of

particular users determines what mechanisms the system must supply to coordinate

users’ actions.

We use the term service for a distinct part of a computer system that manages a

collection of related resources and presents their functionality to users and applications.

For example, we access shared files through a file service; we send documents to

printers through a printing service; we buy goods through an electronic payment service.

The only access we have to the service is via the set of operations that it exports. For

example, a file service provides read, write and delete operations on files.

The fact that services restrict resource access to a well-defined set of operations is

in part standard software engineering practice. But it also reflects the physical

organization of distributed systems. Resources in a distributed system are physically

encapsulated within computers and can only be accessed from other computers by

means of communication. For effective sharing, each resource must be managed by a

program that offers a communication interface enabling the resource to be accessed and

updated reliably and consistently.

The term server is probably familiar to most readers. It refers to a running program

(a process) on a networked computer that accepts requests from programs running on

other computers to perform a service and responds appropriately. The requesting

processes are referred to as clients, and the overall approach is known as client-server

computing. In this approach, requests are sent in messages from clients to a server and

replies are sent in messages from the server to the clients. When the client sends a

request for an operation to be carried out, we say that the client invokes an operation

upon the server. A complete interaction between a client and a server, from the point

when the client sends its request to when it receives the server’s response, is called a

remote invocation.

The same process may be both a client and a server, since servers sometimes

invoke operations on other servers. The terms ‘client’ and ‘server’ apply only to the roles

played in a single request. Clients are active (making requests) and servers are passive

(only waking up when they receive requests); servers run continuously, whereas clients

last only as long as the applications of which they form a part.

Note that while by default the terms ‘client’ and ‘server’ refer to processes rather

than the computers that they execute upon, in everyday parlance those terms also refer

to the computers themselves. Another distinction, which we shall discuss in Chapter 5,32 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

is that in a distributed system written in an object-oriented language, resources may be

encapsulated as objects and accessed by client objects, in which case we speak of a client

object invoking a method upon a server object.

Many, but certainly not all, distributed systems can be constructed entirely in the

form of interacting clients and servers. The World Wide Web, email and networked

printers all fit this model. We discuss alternatives to client-server systems in Chapter 2.

An executing web browser is an example of a client. The web browser

communicates with a web server, to request web pages from it. We consider the Web

and its associated client-server architecture in more detail in Section 1.6.

1.5 Challenges

The examples in Section 1.2 are intended to illustrate the scope of distributed systems

and to suggest the issues that arise in their design. In many of them, significant

challenges were encountered and overcome. As the scope and scale of distributed

systems and applications is extended the same and other challenges are likely to be

encountered. In this section we describe the main challenges.

1.5.1 Heterogeneity

The Internet enables users to access services and run applications over a heterogeneous

collection of computers and networks. Heterogeneity (that is, variety and difference)

applies to all of the following:

• networks;

• computer hardware;

• operating systems;

• programming languages;

• implementations by different developers.

Although the Internet consists of many different sorts of network (illustrated in Figure

1.3), their differences are masked by the fact that all of the computers attached to them

use the Internet protocols to communicate with one another. For example, a computer

attached to an Ethernet has an implementation of the Internet protocols over the

Ethernet, whereas a computer on a different sort of network will need an implementation

of the Internet protocols for that network. Chapter 3 explains how the Internet protocols

are implemented over a variety of different networks.

Data types such as integers may be represented in different ways on different sorts

of hardware – for example, there are two alternatives for the byte ordering of integers.

These differences in representation must be dealt with if messages are to be exchanged

between programs running on different hardware.

Although the operating systems of all computers on the Internet need to include

an implementation of the Internet protocols, they do not necessarily all provide the same

application programming interface to these protocols. For example, the calls for

exchanging messages in UNIX are different from the calls in Windows.SECTION 1.5 CHALLENGES 33

Different programming languages use different representations for characters and

data structures such as arrays and records. These differences must be addressed if

programs written in different languages are to be able to communicate with one another.

Programs written by different developers cannot communicate with one another

unless they use common standards, for example, for network communication and the

representation of primitive data items and data structures in messages. For this to

happen, standards need to be agreed and adopted – as have the Internet protocols.

Middleware • The term middleware applies to a software layer that provides a

programming abstraction as well as masking the heterogeneity of the underlying

networks, hardware, operating systems and programming languages. The Common

Object Request Broker (CORBA), which is described in Chapters 4, 5 and 8, is an

example. Some middleware, such as Java Remote Method Invocation (RMI) (see

Chapter 5), supports only a single programming language. Most middleware is

implemented over the Internet protocols, which themselves mask the differences of the

underlying networks, but all middleware deals with the differences in operating systems

and hardware – how this is done is the main topic of Chapter 4.

In addition to solving the problems of heterogeneity, middleware provides a

uniform computational model for use by the programmers of servers and distributed

applications. Possible models include remote object invocation, remote event

notification, remote SQL access and distributed transaction processing. For example,

CORBA provides remote object invocation, which allows an object in a program

running on one computer to invoke a method of an object in a program running on

another computer. Its implementation hides the fact that messages are passed over a

network in order to send the invocation request and its reply.

Heterogeneity and mobile code • The term mobile code is used to refer to program code

that can be transferred from one computer to another and run at the destination – Java

applets are an example. Code suitable for running on one computer is not necessarily

suitable for running on another because executable programs are normally specific both

to the instruction set and to the host operating system.

The virtual machine approach provides a way of making code executable on a

variety of host computers: the compiler for a particular language generates code for a

virtual machine instead of a particular hardware order code. For example, the Java

compiler produces code for a Java virtual machine, which executes it by interpretation.

The Java virtual machine needs to be implemented once for each type of computer to

enable Java programs to run.

Today, the most commonly used form of mobile code is the inclusion Javascript

programs in some web pages loaded into client browsers. This extension of Web

technology is discussed further in Section 1.6.

1.5.2 Openness

The openness of a computer system is the characteristic that determines whether the

system can be extended and reimplemented in various ways. The openness of distributed

systems is determined primarily by the degree to which new resource-sharing services

can be added and be made available for use by a variety of client programs.34 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

Openness cannot be achieved unless the specification and documentation of the

key software interfaces of the components of a system are made available to software

developers. In a word, the key interfaces are published. This process is akin to the

standardization of interfaces, but it often bypasses official standardization procedures,

which are usually cumbersome and slow-moving.

However, the publication of interfaces is only the starting point for adding and

extending services in a distributed system. The challenge to designers is to tackle the

complexity of distributed systems consisting of many components engineered by

different people.

The designers of the Internet protocols introduced a series of documents called

‘Requests For Comments’, or RFCs, each of which is known by a number. The

specifications of the Internet communication protocols were published in this series in

the early 1980s, followed by specifications for applications that run over them, such as

file transfer, email and telnet by the mid-1980s. This practice has continued and forms

the basis of the technical documentation of the Internet. This series includes discussions

as well as the specifications of protocols. Copies can be obtained from [www.ietf.org].

Thus the publication of the original Internet communication protocols has enabled a

variety of Internet systems and applications including the Web to be built. RFCs are not

the only means of publication. For example, the World Wide Web Consortium (W3C)

develops and publishes standards related to the working of the Web [www.w3.org].

Systems that are designed to support resource sharing in this way are termed open

distributed systems to emphasize the fact that they are extensible. They may be extended

at the hardware level by the addition of computers to the network and at the software

level by the introduction of new services and the reimplementation of old ones, enabling

application programs to share resources. A further benefit that is often cited for open

systems is their independence from individual vendors.

To summarize:

• Open systems are characterized by the fact that their key interfaces are published.

• Open distributed systems are based on the provision of a uniform communication

mechanism and published interfaces for access to shared resources.

• Open distributed systems can be constructed from heterogeneous hardware and

software, possibly from different vendors. But the conformance of each

component to the published standard must be carefully tested and verified if the

system is to work correctly.

1.5.3 Security

Many of the information resources that are made available and maintained in distributed

systems have a high intrinsic value to their users. Their security is therefore of

considerable importance. Security for information resources has three components:

confidentiality (protection against disclosure to unauthorized individuals), integrity

(protection against alteration or corruption), and availability (protection against

interference with the means to access the resources).

Section 1.1 pointed out that although the Internet allows a program in one

computer to communicate with a program in another computer irrespective of itsSECTION 1.5 CHALLENGES 35

location, security risks are associated with allowing free access to all of the resources in

an intranet. Although a firewall can be used to form a barrier around an intranet,

restricting the traffic that can enter and leave, this does not deal with ensuring the

appropriate use of resources by users within an intranet, or with the appropriate use of

resources in the Internet, that are not protected by firewalls.

In a distributed system, clients send requests to access data managed by servers,

which involves sending information in messages over a network. For example:

1. A doctor might request access to hospital patient data or send additions to that data.

2. In electronic commerce and banking, users send their credit card numbers across

the Internet.

In both examples, the challenge is to send sensitive information in a message over a

network in a secure manner. But security is not just a matter of concealing the contents

of messages – it also involves knowing for sure the identity of the user or other agent on

whose behalf a message was sent. In the first example, the server needs to know that the

user is really a doctor, and in the second example, the user needs to be sure of the identity

of the shop or bank with which they are dealing. The second challenge here is to identify

a remote user or other agent correctly. Both of these challenges can be met by the use of

encryption techniques developed for this purpose. They are used widely in the Internet

and are discussed in Chapter 11.

However, the following two security challenges have not yet been fully met:

Denial of service attacks: Another security problem is that a user may wish to

disrupt a service for some reason. This can be achieved by bombarding the service

with such a large number of pointless requests that the serious users are unable to use

it. This is called a denial of service attack. There have been several denial of service

attacks on well-known web services. Currently such attacks are countered by

attempting to catch and punish the perpetrators after the event, but that is not a

general solution to the problem. Countermeasures based on improvements in the

management of networks are under development, and these will be touched on in

Chapter 3.

Security of mobile code: Mobile code needs to be handled with care. Consider

someone who receives an executable program as an electronic mail attachment: the

possible effects of running the program are unpredictable; for example, it may seem

to display an interesting picture but in reality it may access local resources, or perhaps

be part of a denial of service attack. Some measures for securing mobile code are

outlined in Chapter 11.

1.5.4 Scalability

Distributed systems operate effectively and efficiently at many different scales, ranging

from a small intranet to the Internet. A system is described as scalable if it will remain

effective when there is a significant increase in the number of resources and the number

of users. The number of computers and servers in the Internet has increased

dramatically. Figure 1.6 shows the increasing number of computers and web servers

during the 12-year history of the Web up to 2005 [zakon.org]. It is interesting to note the

significant growth in both computers and web servers in this period, but also that the36 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

relative percentage is flattening out – a trend that is explained by the growth of fixed and

mobile personal computing. One web server may also increasingly be hosted on

multiple computers.

The design of scalable distributed systems presents the following challenges:

Controlling the cost of physical resources: As the demand for a resource grows, it

should be possible to extend the system, at reasonable cost, to meet it. For example,

the frequency with which files are accessed in an intranet is likely to grow as the

number of users and computers increases. It must be possible to add server computers

to avoid the performance bottleneck that would arise if a single file server had to

handle all file access requests. In general, for a system with n users to be scalable, the

quantity of physical resources required to support them should be at most O(n) – that

is, proportional to n. For example, if a single file server can support 20 users, then

two such servers should be able to support 40 users. Although that sounds an obvious

goal, it is not necessarily easy to achieve in practice, as we show in Chapter 12.

Controlling the performance loss: Consider the management of a set of data whose

size is proportional to the number of users or resources in the system – for example,

the table with the correspondence between the domain names of computers and their

Internet addresses held by the Domain Name System, which is used mainly to look

up DNS names such as www.amazon.com. Algorithms that use hierarchic structures

scale better than those that use linear structures. But even with hierarchic structures

an increase in size will result in some loss in performance: the time taken to access

hierarchically structured data is O(log n), where n is the size of the set of data. For a

system to be scalable, the maximum performance loss should be no worse than this.

Preventing software resources running out: An example of lack of scalability is

shown by the numbers used as Internet (IP) addresses (computer addresses in the

Internet). In the late 1970s, it was decided to use 32 bits for this purpose, but as will

be explained in Chapter 3, the supply of available Internet addresses is running out.

For this reason, a new version of the protocol with 128-bit Internet addresses is being

adopted, and this will require modifications to many software components. To be fair

Figure 1.6 Growth of the Internet (computers and web servers)

Date Computers Web servers Percentage

1993, July 1,776,000 130 0.008

1995, July 6,642,000 23,500 0.4

1997, July 19,540,000 1,203,096 6

1999, July 56,218,000 6,598,697 12

2001, July 125,888,197 31,299,592 25

2003, July ~200,000,000 42,298,371 21

2005, July 353,284,187 67,571,581 19SECTION 1.5 CHALLENGES 37

to the early designers of the Internet, there is no correct solution to this problem. It is

difficult to predict the demand that will be put on a system years ahead. Moreover,

overcompensating for future growth may be worse than adapting to a change when

we are forced to – larger Internet addresses will occupy extra space in messages and

in computer storage.

Avoiding performance bottlenecks: In general, algorithms should be decentralized

to avoid having performance bottlenecks. We illustrate this point with reference to

the predecessor of the Domain Name System, in which the name table was kept in a

single master file that could be downloaded to any computers that needed it. That was

fine when there were only a few hundred computers in the Internet, but it soon

became a serious performance and administrative bottleneck. The Domain Name

System removed this bottleneck by partitioning the name table between servers

located throughout the Internet and administered locally – see Chapters 3 and 13.

Some shared resources are accessed very frequently; for example, many users

may access the same web page, causing a decline in performance. We shall see in

Chapter 2 that caching and replication may be used to improve the performance of

resources that are very heavily used.

Ideally, the system and application software should not need to change when the scale

of the system increases, but this is difficult to achieve. The issue of scale is a dominant

theme in the development of distributed systems. The techniques that have been

successful are discussed extensively in this book. They include the use of replicated data

(Chapter 18), the associated technique of caching (Chapters 2 and 12) and the

deployment of multiple servers to handle commonly performed tasks, enabling several

similar tasks to be performed concurrently.

1.5.5 Failure handling

Computer systems sometimes fail. When faults occur in hardware or software, programs

may produce incorrect results or may stop before they have completed the intended

computation. We shall discuss and classify a range of possible failure types that can

occur in the processes and networks that comprise a distributed system in Chapter 2.

Failures in a distributed system are partial – that is, some components fail while

others continue to function. Therefore the handling of failures is particularly difficult.

The following techniques for dealing with failures are discussed throughout the book:

Detecting failures: Some failures can be detected. For example, checksums can be

used to detect corrupted data in a message or a file. Chapter 2 explains that it is

difficult or even impossible to detect some other failures, such as a remote crashed

server in the Internet. The challenge is to manage in the presence of failures that

cannot be detected but may be suspected.

Masking failures: Some failures that have been detected can be hidden or made less

severe. Two examples of hiding failures:

1. Messages can be retransmitted when they fail to arrive.

2. File data can be written to a pair of disks so that if one is corrupted, the other may

still be correct.38 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

Just dropping a message that is corrupted is an example of making a fault less severe

– it could be retransmitted. The reader will probably realize that the techniques

described for hiding failures are not guaranteed to work in the worst cases; for

example, the data on the second disk may be corrupted too, or the message may not

get through in a reasonable time however often it is retransmitted.

Tolerating failures: Most of the services in the Internet do exhibit failures – it would

not be practical for them to attempt to detect and hide all of the failures that might

occur in such a large network with so many components. Their clients can be

designed to tolerate failures, which generally involves the users tolerating them as

well. For example, when a web browser cannot contact a web server, it does not make

the user wait for ever while it keeps on trying – it informs the user about the problem,

leaving them free to try again later. Services that tolerate failures are discussed in the

paragraph on redundancy below.

Recovery from failures: Recovery involves the design of software so that the state of

permanent data can be recovered or ‘rolled back’ after a server has crashed. In

general, the computations performed by some programs will be incomplete when a

fault occurs, and the permanent data that they update (files and other material stored

in permanent storage) may not be in a consistent state. Recovery is described in

Chapter 17.

Redundancy: Services can be made to tolerate failures by the use of redundant

components. Consider the following examples:

1. There should always be at least two different routes between any two routers in

the Internet.

2. In the Domain Name System, every name table is replicated in at least two

different servers.

3. A database may be replicated in several servers to ensure that the data remains

accessible after the failure of any single server; the servers can be designed to

detect faults in their peers; when a fault is detected in one server, clients are

redirected to the remaining servers.

The design of effective techniques for keeping replicas of rapidly changing data upto-date without excessive loss of performance is a challenge. Approaches are

discussed in Chapter 18.

Distributed systems provide a high degree of availability in the face of hardware faults.

The availability of a system is a measure of the proportion of time that it is available for

use. When one of the components in a distributed system fails, only the work that was

using the failed component is affected. A user may move to another computer if the one

that they were using fails; a server process can be started on another computer.

1.5.6 Concurrency

Both services and applications provide resources that can be shared by clients in a

distributed system. There is therefore a possibility that several clients will attempt toSECTION 1.5 CHALLENGES 39

access a shared resource at the same time. For example, a data structure that records bids

for an auction may be accessed very frequently when it gets close to the deadline time.

The process that manages a shared resource could take one client request at a time.

But that approach limits throughput. Therefore services and applications generally allow

multiple client requests to be processed concurrently. To make this more concrete,

suppose that each resource is encapsulated as an object and that invocations are executed

in concurrent threads. In this case it is possible that several threads may be executing

concurrently within an object, in which case their operations on the object may conflict

with one another and produce inconsistent results. For example, if two concurrent bids

at an auction are ‘Smith: $122’ and ‘Jones: $111’, and the corresponding operations are

interleaved without any control, then they might get stored as ‘Smith: $111’ and ‘Jones:

$122’.

The moral of this story is that any object that represents a shared resource in a

distributed system must be responsible for ensuring that it operates correctly in a

concurrent environment. This applies not only to servers but also to objects in

applications. Therefore any programmer who takes an implementation of an object that

was not intended for use in a distributed system must do whatever is necessary to make

it safe in a concurrent environment.

For an object to be safe in a concurrent environment, its operations must be

synchronized in such a way that its data remains consistent. This can be achieved by

standard techniques such as semaphores, which are used in most operating systems. This

topic and its extension to collections of distributed shared objects are discussed in

Chapters 7 and 17.

1.5.7 Transparency

Transparency is defined as the concealment from the user and the application

programmer of the separation of components in a distributed system, so that the system

is perceived as a whole rather than as a collection of independent components. The

implications of transparency are a major influence on the design of the system software.

The ANSA Reference Manual [ANSA 1989] and the International Organization

for Standardization’s Reference Model for Open Distributed Processing (RM-ODP)

[ISO 1992] identify eight forms of transparency. We have paraphrased the original

ANSA definitions, replacing their migration transparency with our own mobility

transparency, whose scope is broader:

Access transparency enables local and remote resources to be accessed using

identical operations.

Location transparency enables resources to be accessed without knowledge of their

physical or network location (for example, which building or IP address).

Concurrency transparency enables several processes to operate concurrently using

shared resources without interference between them.40 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

Replication transparency enables multiple instances of resources to be used to

increase reliability and performance without knowledge of the replicas by users or

application programmers.

Failure transparency enables the concealment of faults, allowing users and

application programs to complete their tasks despite the failure of hardware or

software components.

Mobility transparency allows the movement of resources and clients within a system

without affecting the operation of users or programs.

Performance transparency allows the system to be reconfigured to improve

performance as loads vary.

Scaling transparency allows the system and applications to expand in scale without

change to the system structure or the application algorithms.

The two most important transparencies are access and location transparency; their

presence or absence most strongly affects the utilization of distributed resources. They

are sometimes referred to together as network transparency.

As an illustration of access transparency, consider a graphical user interface with

folders, which is the same whether the files inside the folder are local or remote. Another

example is an API for files that uses the same operations to access both local and remote

files (see Chapter 12). As an example of a lack of access transparency, consider a

distributed system that does not allow you to access files on a remote computer unless

you make use of the ftp program to do so.

Web resource names or URLs are location-transparent because the part of the

URL that identifies a web server domain name refers to a computer name in a domain,

rather than to an Internet address. However, URLs are not mobility-transparent, because

someone’s personal web page cannot move to their new place of work in a different

domain – all of the links in other pages will still point to the original page.

In general, identifiers such as URLs that include the domain names of computers

prevent replication transparency. Although the DNS allows a domain name to refer to

several computers, it picks just one of them when it looks up a name. Since a replication

scheme generally needs to be able to access all of the participating computers, it would

need to access each of the DNS entries by name.

As an illustration of the presence of network transparency, consider the use of an

electronic mail address such as Fred.Flintstone@stoneit.com. The address consists of a

user’s name and a domain name. Sending mail to such a user does not involve knowing

their physical or network location. Nor does the procedure to send an email message

depend upon the location of the recipient. Thus electronic mail within the Internet

provides both location and access transparency (that is, network transparency).

Failure transparency can also be illustrated in the context of electronic mail, which

is eventually delivered, even when servers or communication links fail. The faults are

masked by attempting to retransmit messages until they are successfully delivered, even

if it takes several days. Middleware generally converts the failures of networks and

processes into programming-level exceptions (see Chapter 5 for an explanation).

To illustrate mobility transparency, consider the case of mobile phones. Suppose

that both caller and callee are travelling by train in different parts of a country, movingSECTION 1.5 CHALLENGES 41

from one environment (cell) to another. We regard the caller’s phone as the client and

the callee’s phone as a resource. The two phone users making the call are unaware of the

mobility of the phones (the client and the resource) between cells.

Transparency hides and renders anonymous the resources that are not of direct

relevance to the task in hand for users and application programmers. For example, it is

generally desirable for similar hardware resources to be allocated interchangeably to

perform a task – the identity of a processor used to execute a process is generally hidden

from the user and remains anonymous. As pointed out in Section 1.3.2, this may not

always be what is required: for example, a traveller who attaches a laptop computer to

the local network in each office visited should make use of local services such as the

send mail service, using different servers at each location. Even within a building, it is

normal to arrange for a document to be printed at a particular, named printer: usually one

that is near to the user.

1.5.8 Quality of service

Once users are provided with the functionality that they require of a service, such as the

file service in a distributed system, we can go on to ask about the quality of the service

provided. The main nonfunctional properties of systems that affect the quality of the

service experienced by clients and users are reliability, security and performance.

Adaptability to meet changing system configurations and resource availability has been

recognized as a further important aspect of service quality.

Reliability and security issues are critical in the design of most computer systems.

The performance aspect of quality of service was originally defined in terms of

responsiveness and computational throughput, but it has been redefined in terms of

ability to meet timeliness guarantees, as discussed in the following paragraphs.

Some applications, including multimedia applications, handle time-critical data –

streams of data that are required to be processed or transferred from one process to

another at a fixed rate. For example, a movie service might consist of a client program

that is retrieving a film from a video server and presenting it on the user’s screen. For a

satisfactory result the successive frames of video need to be displayed to the user within

some specified time limits.

In fact, the abbreviation QoS has effectively been commandeered to refer to the

ability of systems to meet such deadlines. Its achievement depends upon the availability

of the necessary computing and network resources at the appropriate times. This implies

a requirement for the system to provide guaranteed computing and communication

resources that are sufficient to enable applications to complete each task on time (for

example, the task of displaying a frame of video).

The networks commonly used today have high performance – for example, BBC

iPlayer generally performs acceptably – but when networks are heavily loaded their

performance can deteriorate, and no guarantees are provided. QoS applies to operating

systems as well as networks. Each critical resource must be reserved by the applications

that require QoS, and there must be resource managers that provide guarantees.

Reservation requests that cannot be met are rejected. These issues will be addressed

further in Chapter 20.42 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

1.6 Case study: The World Wide Web

The World Wide Web [www.w3.org I, Berners-Lee 1991] is an evolving system for

publishing and accessing resources and services across the Internet. Through commonly

available web browsers, users retrieve and view documents of many types, listen to

audio streams and view video streams, and interact with an unlimited set of services.

The Web began life at the European centre for nuclear research (CERN),

Switzerland, in 1989 as a vehicle for exchanging documents between a community of

physicists connected by the Internet [Berners-Lee 1999]. A key feature of the Web is that

it provides a hypertext structure among the documents that it stores, reflecting the users’

requirement to organize their knowledge. This means that documents contain links (or

hyperlinks) – references to other documents and resources that are also stored in the Web.

It is fundamental to the user’s experience of the Web that when they encounter a given

image or piece of text within a document, this will frequently be accompanied by links to

related documents and other resources. The structure of links can be arbitrarily complex and

the set of resources that can be added is unlimited – the ‘web’ of links is indeed world-wide.

Bush [1945] conceived of hypertextual structures over 50 years ago; it was with the

development of the Internet that this idea could be manifested on a world-wide scale.

The Web is an open system: it can be extended and implemented in new ways

without disturbing its existing functionality (see Section 1.5.2). First, its operation is

based on communication standards and document or content standards that are freely

published and widely implemented. For example, there are many types of browser, each

in many cases implemented on several platforms; and there are many implementations

of web servers. Any conformant browser can retrieve resources from any conformant

server. So users have access to browsers on the majority of the devices that they use,

from mobile phones to desktop computers.

Second, the Web is open with respect to the types of resource that can be published

and shared on it. At its simplest, a resource on the Web is a web page or some other type

of content that can be presented to the user, such as media files and documents in

Portable Document Format. If somebody invents, say, a new image-storage format, then

images in this format can immediately be published on the Web. Users require a means

of viewing images in this new format, but browsers are designed to accommodate new

content-presentation functionality in the form of ‘helper’ applications and ‘plug-ins’.

The Web has moved beyond these simple data resources to encompass services,

such as electronic purchasing of goods. It has evolved without changing its basic

architecture. The Web is based on three main standard technological components:

• the HyperText Markup Language (HTML), a language for specifying the contents

and layout of pages as they are displayed by web browsers;

• Uniform Resource Locators (URLs), also known as Uniform Resource Identifiers

(URIs), which identify documents and other resources stored as part of the Web;

• a client-server system architecture, with standard rules for interaction (the

HyperText Transfer Protocol – HTTP) by which browsers and other clients fetch

documents and other resources from web servers. Figure 1.7 shows some web

servers, and browsers making requests to them. It is an important feature that users

may locate and manage their own web servers anywhere on the Internet.SECTION 1.6 CASE STUDY: THE WORLD WIDE WEB 43

We now discuss these components in turn, and in so doing explain the operation of

browsers and web servers when a user fetches web pages and clicks on the links within

them.

HTML • The HyperText Markup Language [www.w3.org II] is used to specify the text

and images that make up the contents of a web page, and to specify how they are laid

out and formatted for presentation to the user. A web page contains such structured items

as headings, paragraphs, tables and images. HTML is also used to specify links and

which resources are associated with them.

Users may produce HTML by hand, using a standard text editor, but they more

commonly use an HTML-aware ‘wysiwyg’ editor that generates HTML from a layout

that they create graphically. A typical piece of HTML text follows:

<IMG SRC = “http://www.cdk5.net/WebExample/Images/earth.jpg”> 1

<P> 2

Welcome to Earth! Visitors may also be interested in taking a look at the 3

<A HREF = “http://www.cdk5.net/WebExample/moon.html”>Moon</A>. 4

</P> 5

This HTML text is stored in a file that a web server can access – let us say the file

earth.html. A browser retrieves the contents of this file from a web server – in this case

a server on a computer called www.cdk5.net. The browser reads the content returned by

the server and renders it into formatted text and images laid out on a web page in the

familiar fashion. Only the browser – not the server – interprets the HTML text. But the

server does inform the browser of the type of content it is returning, to distinguish it

from, say, a document in Portable Document Format. The server can infer the content

type from the filename extension ‘.html’.

Note that the HTML directives, known as tags, are enclosed by angle brackets,

such as <P>. Line 1 of the example identifies a file containing an image for

presentation. Its URL is http://www.cdk5.net/WebExample/Images/earth.jpg. Lines 2

and 5 are directives to begin and end a paragraph, respectively. Lines 3 and 4 contain

text to be displayed on the web page in the standard paragraph format.

Figure 1.7 Web servers and web browsers

Internet

Web servers Browsers

www.google.com

www.cdk5.net

www.w3c.org

standards

faq.html

http://www.google.com/search?q=obama

GET http://www.cdk5.net

File system of

www.w3c.org

http://www.w3.org/standards/faq.html#conformance44 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

Line 4 specifies a link in the web page. It contains the word ‘Moon’ surrounded

by two related HTML tags, <A HREF...> and </A>. The text between these tags is what

appears in the link as it is presented on the web page. Most browsers are configured to

show the text of links underlined by default, so what the user will see in that paragraph

is:

Welcome to Earth! Visitors may also be interested in taking a look at the Moon.

The browser records the association between the link’s displayed text and the URL

contained in the <A HREF...> tag – in this case:

http://www.cdk5.net/WebExample/moon.html

When the user clicks on the text, the browser retrieves the resource identified by the

corresponding URL and presents it to the user. In the example, the resource is an HTML

file specifying a web page about the Moon.

URLs • The purpose of a Uniform Resource Locator [www.w3.org III] is to identify a

resource. Indeed, the term used in web architecture documents is Uniform Resource

Identifier (URI), but in this book the better-known term URL will be used when no

confusion can arise. Browsers examine URLs in order to access the corresponding

resources. Sometimes the user types a URL into the browser. More commonly, the

browser looks up the corresponding URL when the user clicks on a link or selects one

of their ‘bookmarks’; or when the browser fetches a resource embedded in a web page,

such as an image.

Every URL, in its full, absolute form, has two top-level components:

scheme : scheme-specific-identifier

The first component, the ‘scheme’, declares which type of URL this is. URLs are

required to identify a variety of resources. For example, mailto:joe@anISP.net

identifies a user’s email address; ftp://ftp.downloadIt.com/software/aProg.exe identifies

a file that is to be retrieved using the File Transfer Protocol (FTP) rather than the more

commonly used protocol HTTP. Other examples of schemes are ‘tel’ (used to specify a

telephone number to dial, which is particularly useful when browsing on a mobile

phone) and ‘tag’ (used to identify an arbitrary entity).

The Web is open with respect to the types of resources it can be used to access, by

virtue of the scheme designators in URLs. If somebody invents a useful new type of

‘widget’ resource – perhaps with its own addressing scheme for locating widgets and its

own protocol for accessing them – then the world can start using URLs of the form

widget:.... Of course, browsers must be given the capability to use the new ‘widget’

protocol, but this can be done by adding a plug-in.

HTTP URLs are the most widely used, for accessing resources using the standard

HTTP protocol. An HTTP URL has two main jobs: to identify which web server

maintains the resource, and to identify which of the resources at that server is required.

Figure 1.7 shows three browsers issuing requests for resources managed by three web

servers. The topmost browser is issuing a query to a search engine. The middle browser

requires the default page of another web site. The bottommost browser requires a web

page that is specified in full, including a path name relative to the server. The files for a

given web server are maintained in one or more subtrees (directories) of the server’s file

system, and each resource is identified by a path name relative to the server.SECTION 1.6 CASE STUDY: THE WORLD WIDE WEB 45

In general, HTTP URLs are of the following form:

http:// servername [:port] [/pathName] [?query] [ #fragment]

where items in square brackets are optional. A full HTTP URL always begins with the

string ‘http://’ followed by a server name, expressed as a Domain Name System (DNS)

name (see Section 13.2). The server’s DNS name is optionally followed by the number

of the ‘port’ on which the server listens for requests (see Chapter 4), which is 80 by

default. Then comes an optional path name of the server’s resource. If this is absent then

the server’s default web page is required. Finally, the URL optionally ends in a query

component – for example, when a user submits the entries in a form such as a search

engine’s query page – and/or a fragment identifier, which identifies a component of the

resource.

Consider the URLs:

http://www.cdk5.net

http://www.w3.org/standards/faq.html#conformance

http://www.google.com/search?q=obama

These can be broken down as follows:

The first URL designates the default page supplied by www.cdk5.net. The next identifies

a fragment of an HTML file whose path name is standards/faq.html relative to the server

www.w3.org. The fragment’s identifier (specified after the ‘#’ character in the URL) is

intro, and a browser will search for that fragment identifier within the HTML text after

it has downloaded the whole file. The third URL specifies a query to a search engine.

The path identifies a program called ‘search’, and the string after the ‘?’ character

encodes a query string supplied as arguments to this program. We discuss URLs that

identify programmatic resources in more detail when we consider more advanced

features below.

Publishing a resource: While the Web has a clearly defined model for accessing a

resource from its URL, the exact methods for publishing resources on the Web are

dependent upon the web server implementation. In terms of low-level mechanisms, the

simplest method of publishing a resource on the Web is to place the corresponding file

in a directory that the web server can access. Knowing the name of the server S and a

path name for the file P that the server can recognize, the user then constructs the URL

as http://S/P. The user puts this URL in a link from an existing document or distributes

the URL to other users, for example by email.

It is common for such concerns to be hidden from users when they generate

content. For example, ‘bloggers’ typically use software tools, themselves implemented

as web pages, to create organized collections of journal pages. Product pages for a

company’s web site are typically created using a content management system, again by

Server DNS name Path name Query Fragment

www.cdk5.net (default) (none) (none)

www.w3.org standards/faq.html (none) intro

www.google.com search q=obama (none)46 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

directly interacting with the web site through administrative web pages. The database or

file system on which the product pages are based is transparent.

Finally, Huang et al. [2000] provide a model for inserting content into the Web

with minimal human intervention. This is particularly relevant where users need to

extract content from a variety of devices, such as cameras, for publication in web pages.

HTTP • The HyperText Transfer Protocol [www.w3.org IV] defines the ways in which

browsers and other types of client interact with web servers. Chapter 5 will consider

HTTP in more detail, but here we outline its main features (restricting our discussion to

the retrieval of resources in files):

Request-reply interactions: HTTP is a ‘request-reply’ protocol. The client sends a

request message to the server containing the URL of the required resource. The

server looks up the path name and, if it exists, sends back the resource’s content in a

reply message to the client. Otherwise, it sends back an error response such as the

familiar ‘404 Not Found’. HTTP defines a small set of operations or methods that can

be performed on a resource. The most common are GET, to retrieve data from the

resource, and POST, to provide data to the resource.

Content types: Browsers are not necessarily capable of handling every type of

content. When a browser makes a request, it includes a list of the types of content it

prefers – for example, in principle it may be able to display images in ‘GIF’ format

but not ‘JPEG’ format. The server may be able to take this into account when it

returns content to the browser. The server includes the content type in the reply

message so that the browser will know how to process it. The strings that denote the

type of content are called MIME types, and they are standardized in RFC 1521 [Freed

and Borenstein 1996]. For example, if the content is of type ‘text/html’ then a

browser will interpret the text as HTML and display it; if the content is of type

‘image/GIF’ then the browser will render it as an image in ‘GIF’ format; if the

content type is ‘application/zip’ then it is data compressed in ‘zip’ format, and the

browser will launch an external helper application to decompress it. The set of

actions that a browser will take for a given type of content is configurable, and

readers may care to check these settings for their own browsers.

One resource per request: Clients specify one resource per HTTP request. If a web

page contains nine images, say, then the browser will issue a total of ten separate

requests to obtain the entire contents of the page. Browsers typically make several

requests concurrently, to reduce the overall delay to the user.

Simple access control: By default, any user with network connectivity to a web

server can access any of its published resources. If users wish to restrict access to a

resource, then they can configure the server to issue a ‘challenge’ to any client that

requests it. The corresponding user then has to prove that they have the right to access

the resource, for example, by typing in a password.

Dynamic pages • So far we have described how users can publish web pages and other

content stored in files on the Web. However, much of the users’ experience of the Web

is that of interacting with services rather than retrieving data. For example, when

purchasing an item at an online store, the user often fills out a web form to provide

personal details or to specify exactly what they wish to purchase. A web form is a webSECTION 1.6 CASE STUDY: THE WORLD WIDE WEB 47

page containing instructions for the user and input widgets such as text fields and check

boxes. When the user submits the form (usually by pressing a button or the ‘return’ key),

the browser sends an HTTP request to a web server, containing the values that the user

has entered.

Since the result of the request depends upon the user’s input, the server has to

process the user’s input. Therefore the URL or its initial component designates a

program on the server, not a file. If the user’s input is a reasonably small set of

parameters it is often sent as the query component of the URL, using the GET

method; alternatively, it is sent as additional data in the request using the POST

method. For example, a request containing the following URL invokes a program

called ‘search’ at www.google.com and specifies a query string of ‘obama’:

http://www.google.com/search?q=obama.

That ‘search’ program produces HTML text as its output, and the user will see a

listing of pages that contain the word ‘obama’. (The reader may care to enter a query

into their favourite search engine and notice the URL that the browser displays when the

result is returned.) The server returns the HTML text that the program generates just as

though it had retrieved it from a file. In other words, the difference between static

content fetched from a file and content that is dynamically generated is transparent to

the browser.

A program that web servers run to generate content for their clients is referred to

as a Common Gateway Interface (CGI) program. A CGI program may have any

application-specific functionality, as long as it can parse the arguments that the client

provides to it and produce content of the required type (usually HTML text). The

program will often consult or update a database in processing the request.

Downloaded code: A CGI program runs at the server. Sometimes the designers of web

services require some service-related code to run inside the browser, at the user’s

computer. In particular, code written in Javascript [www.netscape.com] is often

downloaded with a web page containing a form, in order to provide better-quality

interaction with the user than that supported by HTML’s standard widgets. A Javascriptenhanced page can give the user immediate feedback on invalid entries, instead of

forcing the user to check the values at the server, which would take much longer.

Javascript can also be used to update parts of a web page’s contents without

fetching an entirely new version of the page and re-rendering it. These dynamic updates

occur either due to a user action (such as clicking on a link or a radio button), or when

the browser acquires new data from the server that supplied the web page. In the latter

case, since the timing of the data’s arrival is unconnected with any user action at the

browser itself, it is termed asynchronous. A technique known as AJAX (Asynchronous

Javascript And XML) is used in such cases. AJAX is described more fully in Section

2.3.2.

An alternative to a Javascript program is an applet: an application written in the

Java language [Flanagan 2002], which the browser automatically downloads and runs

when it fetches a corresponding web page. Applets may access the network and provide

customized user interfaces. For example, ‘chat’ applications are sometimes

implemented as applets that run on the users’ browsers, together with a server program.

The applets send the users’ text to the server, which in turn distributes it to all the applets

for presentation to the user. We discuss applets in more detail in Section 2.3.1.48 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

Web services • So far we have discussed the Web largely from the point of view of a

user operating a browser. But programs other than browsers can be clients of the Web,

too; indeed, programmatic access to web resources is commonplace.

However, HTML is inadequate for programmatic interoperation. There is an

increasing need to exchange many types of structured data on the Web, but HTML is

limited in that it is not extensible to applications beyond information browsing. HTML

has a static set of structures such as paragraphs, and they are bound up with the way that

the data is to be presented to users. The Extensible Markup Language (XML) (see

Section 4.3.3) has been designed as a way of representing data in standard, structured,

application-specific forms. In principle, data expressed in XML is portable between

applications since it is self-describing: it contains the names, types and structure of the

data elements within it. For example, XML may be used to describe products or

information about users, for many different services or applications. In the HTTP

protocol, XML data can be transmitted by the POST and GET operations. In AJAX it

can be used to provide data to Javascript programs in browsers.

Web resources provide service-specific operations. For example, in the store at

amazon.com, web service operations include one to order a book and another to check

the current status of an order. As we have mentioned, HTTP provides a small set of

operations that are applicable to any resource. These include principally the GET and

POST methods on existing resources, and the PUT and DELETE operations,

respectively. for creating and deleting web resources. Any operation on a resource can

be invoked using one of the GET or POST methods, with structured content used to

specify the operation’s parameters, results and error responses. The so-called REST

(REpresentational State Transfer) architecture for web services [Fielding 2000] adopts

this approach on the basis of its extensibility: every resource on the Web has a URL and

responds to the same set of operations, although the processing of the operations can

vary widely from resource to resource. The flip-side of that extensibility can be a lack

of robustness in how software operates. Chapter 9 further describes REST and takes an

in-depth look at the web services framework, which enables the designers of web

services to describe to programmers more specifically what service-specific operations

are available and how clients must access them.

Discussion of the Web • The Web’s phenomenal success rests upon the relative ease

with which many individual and organizational sources can publish resources, the

suitability of its hypertext structure for organizing many types of information, and the

openness of its system architecture. The standards upon which its architecture is based

are simple and they were widely published at an early stage. They have enabled many

new types of resources and services to be integrated.

The Web’s success belies some design problems. First, its hypertext model is

lacking in some respects. If a resource is deleted or moved, so-called ‘dangling’ links to

that resource may still remain, causing frustration for users. And there is the familiar

problem of users getting ‘lost in hyperspace’. Users often find themselves confused,

following many disparate links, referencing pages from a disparate collection of

sources, and of dubious reliability in some cases.

Search engines are a highly popular alternative to following links as a means of

finding information on the Web, but these are imperfect at producing what the user

specifically intends. One approach to this problem, exemplified in the ResourceSECTION 1.7 SUMMARY 49

Description Framework [www.w3.org V], is to produce standard vocabularies, syntax

and semantics for expressing metadata about the things in our world, and to encapsulate

that metadata in corresponding web resources for programmatic access. Rather than

searching for words that occur in web pages, programs can then, in principle, perform

searches against the metadata to compile lists of related links based on semantic

matching. Collectively, the web of linked metadata resources is what is meant by the

semantic web.

As a system architecture the Web faces problems of scale. Popular web servers

may experience many ‘hits’ per second, and as a result the response to users can be slow.

Chapter 2 describes the use of caching in browsers and proxy servers to increase

responsiveness, and the division of the server’s load across clusters of computers.

1.7 Summary

Distributed systems are everywhere. The Internet enables users throughout the world to

access its services wherever they may be located. Each organization manages an

intranet, which provides local services and Internet services for local users and generally

provides services to other users in the Internet. Small distributed systems can be

constructed from mobile computers and other small computational devices that are

attached to a wireless network.

Resource sharing is the main motivating factor for constructing distributed

systems. Resources such as printers, files, web pages or database records are managed

by servers of the appropriate type. For example, web servers manage web pages and

other web resources. Resources are accessed by clients – for example, the clients of web

servers are generally called browsers.

The construction of distributed systems produces many challenges:

Heterogeneity: They must be constructed from a variety of different networks,

operating systems, computer hardware and programming languages. The Internet

communication protocols mask the difference in networks, and middleware can deal

with the other differences.

Openness: Distributed systems should be extensible – the first step is to publish the

interfaces of the components, but the integration of components written by different

programmers is a real challenge.

Security: Encryption can be used to provide adequate protection of shared resources

and to keep sensitive information secret when it is transmitted in messages over a

network. Denial of service attacks are still a problem.

Scalability: A distributed system is scalable if the cost of adding a user is a constant

amount in terms of the resources that must be added. The algorithms used to access

shared data should avoid performance bottlenecks and data should be structured

hierarchically to get the best access times. Frequently accessed data can be replicated.

Failure handling: Any process, computer or network may fail independently of the

others. Therefore each component needs to be aware of the possible ways in which50 CHAPTER 1 CHARACTERIZATION OF DISTRIBUTED SYSTEMS

the components it depends on may fail and be designed to deal with each of those

failures appropriately.

Concurrency: The presence of multiple users in a distributed system is a source of

concurrent requests to its resources. Each resource must be designed to be safe in a

concurrent environment.

Transparency: The aim is to make certain aspects of distribution invisible to the

application programmer so that they need only be concerned with the design of their

particular application. For example, they need not be concerned with its location or

the details of how its operations are accessed by other components, or whether it will

be replicated or migrated. Even failures of networks and processes can be presented

to application programmers in the form of exceptions – but they must be handled.

Quality of service. It is not sufficient to provide access to services in distributed

systems. In particular, it is also important to provide guarantees regarding the

qualities associated with such service access. Examples of such qualities include

parameters related to performance, security and reliability.

EXERCISES

1.1 We define a distributed system as one in which hardware or software components located at

networked computers communicate and coordinate their actions only by passing messages.

What are the consequences of defining a distributed system in this manner? page 18

1.2 How might the clocks in two computers that are linked by a local network be

synchronized without reference to an external time source? What factors limit the

accuracy of the procedure you have described? How could the clocks in a large number

of computers connected by the Internet be synchronized? Discuss the accuracy of that

procedure. page 18

1.3 Consider the implementation strategies for massively multiplayer online games as

discussed in Section 1.2.2. In particular, what advantages do you see in adopting a single

server approach for representing the state of the multiplayer game? What problems can

you identify and how might they be resolved? page 21

1.4 A user arrives at a railway station that they has never visited before, carrying a PDA that

is capable of wireless networking. Suggest how the user could be provided with

information about the local services and amenities at that station, without entering the

station’s name or attributes. What technical challenges must be overcome? page 29

1.5 Distributed systems are going through a period of significant change, which can be traced

back to a number of influential trends. Can you identify what these might be? page 24

1.6 Due to the increasing maturity of distributed systems infrastructure, organizations are moving

towards viewing distributed systems as a utility. In this model, resources are provided by

appropriate service suppliers and effectively rented rather than owned by the end user. Explain

this model with respect to physical resources and software services. Can you give examples

of some companies that support such software services? page 29EXERCISES 51

1.7 A server program written in one language (for example, C++) provides the

implementation of a BLOB object that is intended to be accessed by clients that may be

written in a different language (for example, Java). The client and server computers may

have different hardware, but all of them are attached to an internet. Describe the

problems due to each of the five aspects of heterogeneity that need to be solved to make

it possible for a client object to invoke a method on the server object. page 32

1.8 What is client-server computing? Which of these roles is an active role, and which is a

passive one? Explain remote invocation in this context. page 31

1.9 Suppose that the operations of the BLOB object are separated into two categories –

public operations that are available to all users and protected operations that are

available only to certain named users. State all of the problems involved in ensuring that

only the named users can use a protected operation. Supposing that access to a protected

operation provides information that should not be revealed to all users, what further

problems arise? page 34

1.10 The INFO service manages a potentially very large set of resources, each of which can

be accessed by users throughout the Internet by means of a key (a string name). Discuss

an approach to the design of the names of the resources that achieves the minimum loss

of performance as the number of resources in the service increases. Suggest how the

INFO service can be implemented so as to avoid performance bottlenecks when the

number of users becomes very large. page 35

1.11 A distributed system is described as scalable if it remains effective when there is a

significant increase in the number of resources and the number of users. However, these

systems sometimes face performance bottlenecks. How can these be avoided? page 37

1.12 A server process maintains a shared information object such as the BLOB object of

Exercise 1.7. Give arguments for and against allowing the client requests to be executed

concurrently by the server. In the case that they are executed concurrently, give an

example of possible ‘interference’ that can occur between the operations of different

clients. Suggest how such interference may be prevented. page 38

1.13 The ANSA Reference Manual [ANSA 1989] identified eight forms of transparency

in a distributed system. Which are the two most important transparencies among

these? page 40

1.14 One of the main standard technological components of the Web is the HyperText

Transfer Protocol (HTTP), which defines the ways in which browsers and other types of

clients interact with web servers. What are the different features of the HTTP? page 46

1.15 What are the two main functions of an HTTP URL? Explain its general form. Identify

the server DNS name, path name, query, and fragment for the URL http://www.google.com/

search?q=sabretooth. page 44This page intentionally left blank53

2

SYSTEM MODELS

2.1 Introduction

2.2 Physical models

2.3 Architectural models

2.4 Fundamental models

2.5 Summary

This chapter provides an explanation of three important and complementary ways in

which the design of distributed systems can usefully be described and discussed:

Physical models consider the types of computers and devices that constitute a system

and their interconnectivity, without details of specific technologies.

Architectural models describe a system in terms of the computational and

communication tasks performed by its computational elements; the computational

elements being individual computers or aggregates of them supported by appropriate

network interconnections. Client-server and peer-to-peer are two of the most

commonly used forms of architectural model for distributed systems.

Fundamental models take an abstract perspective in order to describe solutions to

individual issues faced by most distributed systems.

There is no global time in a distributed system, so the clocks on different computers do

not necessarily give the same time as one another. All communication between processes

is achieved by means of messages. Message communication over a computer network

can be affected by delays, can suffer from a variety of failures and is vulnerable to security

attacks. These issues are addressed by three models:

• The interaction model deals with performance and with the difficulty of setting time

limits in a distributed system, for example for message delivery.

• The failure model attempts to give a precise specification of the faults that can be

exhibited by processes and communication channels. It defines reliable

communication and correct processes.

• The security model discusses the possible threats to processes and communication

channels. It introduces the concept of a secure channel, which is secure against

those threats.54 CHAPTER 2 SYSTEM MODELS

2.1 Introduction

Systems that are intended for use in real-world environments should be designed to

function correctly in the widest possible range of circumstances and in the face of many

possible difficulties and threats (for some examples, see the box at the bottom of this

page). The discussion and examples of Chapter 1 suggest that distributed systems of

different types share important underlying properties and give rise to common design

problems. In this chapter we show how the properties and design issues of distributed

systems can be captured and discussed through the use of descriptive models. Each type

of model is intended to provide an abstract, simplified but consistent description of a

relevant aspect of distributed system design:

Physical models are the most explicit way in which to describe a system; they

capture the hardware composition of a system in terms of the computers (and other

devices, such as mobile phones) and their interconnecting networks.

Architectural models describe a system in terms of the computational and

communication tasks performed by its computational elements; the computational

elements being individual computers or aggregates of them supported by appropriate

network interconnections.

Fundamental models take an abstract perspective in order to examine individual

aspects of a distributed system. In this chapter we introduce fundamental models that

examine three important aspects of distributed systems: interaction models, which

consider the structure and sequencing of the communication between the elements of

the system; failure models, which consider the ways in which a system may fail to

operate correctly and; security models, which consider how the system is protected

against attempts to interfere with its correct operation or to steal its data.

Difficulties and threats for distributed systems • Here are some of the problems that

the designers of distributed systems face.

Widely varying modes of use: The component parts of systems are subject to wide

variations in workload – for example, some web pages are accessed several million

times a day. Some parts of a system may be disconnected, or poorly connected some

of the time – for example, when mobile computers are included in a system. Some

applications have special requirements for high communication bandwidth and low

latency – for example, multimedia applications.

Wide range of system environments: A distributed system must accommodate

heterogeneous hardware, operating systems and networks. The networks may differ

widely in performance – wireless networks operate at a fraction of the speed of local

networks. Systems of widely differing scales, ranging from tens of computers to

millions of computers, must be supported.

Internal problems: Non-synchronized clocks, conflicting data updates and many

modes of hardware and software failure involving the individual system components.

External threats: Attacks on data integrity and secrecy, denial of service attacks.SECTION 2.2 PHYSICAL MODELS 55

2.2 Physical models

A physical model is a representation of the underlying hardware elements of a

distributed system that abstracts away from specific details of the computer and

networking technologies employed.

Baseline physical model: A distributed system was defined in Chapter 1 as one in which

hardware or software components located at networked computers communicate and

coordinate their actions only by passing messages. This leads to a minimal physical

model of a distributed system as an extensible set of computer nodes interconnected by

a computer network for the required passing of messages.

Beyond this baseline model, we can usefully identify three generations of distributed

systems.

Early distributed systems: Such systems emerged in the late 1970s and early 1980s in

response to the emergence of local area networking technology, usually Ethernet (see

Section 3.5). These systems typically consisted of between 10 and 100 nodes

interconnected by a local area network, with limited Internet connectivity and supported

a small range of services such as shared local printers and file servers as well as email

and file transfer across the Internet. Individual systems were largely homogeneous and

openness was not a primary concern. Providing quality of service was still very much in

its infancy and was a focal point for much of the research around such early systems.

Internet-scale distributed systems: Building on this foundation, larger-scale distributed

systems started to emerge in the 1990s in response to the dramatic growth of the Internet

during this time (for example, the Google search engine was first launched in 1996). In

such systems, the underlying physical infrastructure consists of a physical model as

illustrated in Chapter 1, Figure 1.3; that is, an extensible set of nodes interconnected by

a network of networks (the Internet). Such systems exploit the infrastructure offered by

the Internet to become truly global. They incorporate large numbers of nodes and

provide distributed system services for global organizations and across organizational

boundaries. The level of heterogeneity in such systems is significant in terms of

networks, computer architecture, operating systems, languages employed and the

development teams involved. This has led to an increasing emphasis on open standards

and associated middleware technologies such as CORBA and more recently, web

services. Additional services were employed to provide end-to-end quality of service

properties in such global systems.

Contemporary distributed systems: In the above systems, nodes were typically desktop

computers and therefore relatively static (that is, remaining in one physical location for

extended periods), discrete (not embedded within other physical entities) and

autonomous (to a large extent independent of other computers in terms of their physical

infrastructure). The key trends identified in Section 1.3 have resulted in significant

further developments in physical models:

• The emergence of mobile computing has led to physical models where nodes such

as laptops or smart phones may move from location to location in a distributed

system, leading to the need for added capabilities such as service discovery and

support for spontaneous interoperation.56 CHAPTER 2 SYSTEM MODELS

• The emergence of ubiquitous computing has led to a move from discrete nodes to

architectures where computers are embedded in everyday objects and in the

surrounding environment (for example, in washing machines or in smart homes

more generally).

• The emergence of cloud computing and, in particular, cluster architectures has led

to a move from autonomous nodes performing a given role to pools of nodes that

together provide a given service (for example, a search service as offered by

Google).

The end result is a physical architecture with a significant increase in the level of

heterogeneity embracing, for example, the tiniest embedded devices utilized in

ubiquitous computing through to complex computational elements found in Grid

computing. These systems deploy an increasingly varied set of networking technologies

and offer a wide variety of applications and services. Such systems potentially involve

up to hundreds of thousands of nodes.

Distributed systems of systems • A recent report discusses the emergence of ultralarge-scale (ULS) distributed systems [www.sei.cmu.edu]. The report captures the

complexity of modern distributed systems by referring to such (physical) architectures

as systems of systems (mirroring the view of the Internet as a network of networks). A

system of systems can be defined as a complex system consisting of a series of

subsystems that are systems in their own right and that come together to perform a

particular task or tasks.

As an example of a system of systems, consider an environmental management

system for flood prediction. In such a scenario, there will be sensor networks deployed

to monitor the state of various environmental parameters relating to rivers, flood plains,

tidal effects and so on. This can then be coupled with systems that are responsible for

predicting the likelihood of floods, by running (often complex) simulations on, for

example, cluster computers (as discussed in Chapter 1). Other systems may be

established to maintain and analyze historical data or to provide early warning systems

to key stakeholders via mobile phones.

Summary • The overall historical development captured in this section is summarized

in Figure 2.1, with the table highlighting the significant challenges associated with

contemporary distributed systems in terms of managing the levels of heterogeneity and

providing key properties such as openness and quality of service.

2.3 Architectural models

The architecture of a system is its structure in terms of separately specified components

and their interrelationships. The overall goal is to ensure that the structure will meet

present and likely future demands on it. Major concerns are to make the system reliable,

manageable, adaptable and cost-effective. The architectural design of a building has

similar aspects – it determines not only its appearance but also its general structure and

architectural style (gothic, neo-classical, modern) and provides a consistent frame of

reference for the design.SECTION 2.3 ARCHITECTURAL MODELS 57

In this section we describe the main architectural models employed in distributed

systems – the architectural styles of distributed systems. In particular, we lay the

groundwork for a thorough understanding of approaches such as client-server models,

peer-to-peer approaches, distributed objects, distributed components, distributed eventbased systems and the key differences between these styles.

The section adopts a three-stage approach:

• looking at the core underlying architectural elements that underpin modern

distributed systems, highlighting the diversity of approaches that now exist;

• examining composite architectural patterns that can be used in isolation or, more

commonly, in combination, in developing more sophisticated distributed systems

solutions;

• and finally, considering middleware platforms that are available to support the

various styles of programming that emerge from the above architectural styles.

Note that there are many trade-offs associated with the choices identified in this chapter

in terms of the architectural elements employed, the patterns adopted and (where

appropriate) the middleware used, for example affecting the performance and

effectiveness of the resulting system. Understanding such trade-offs is arguably the key

skill in distributed systems design.

2.3.1 Architectural elements

To understand the fundamental building blocks of a distributed system, it is necessary

to consider four key questions:

• What are the entities that are communicating in the distributed system?

Figure 2.1 Generations of distributed systems

Distributed systems: Early Internet-scale Contemporary

Scale Small Large Ultra-large

Heterogeneity

Limited (typically

relatively homogenous

configurations)

Significant in terms of

platforms, languages

and middleware

Added dimensions

introduced including

radically different styles of

architecture

Openness

Not a priority

Significant priority

with range of standards

introduced

Major research challenge

with existing standards not

yet able to embrace

complex systems

Quality of service

In its infancy

Significant priority

with range of services

introduced

Major research challenge

with existing services not

yet able to embrace

complex systems58 CHAPTER 2 SYSTEM MODELS

• How do they communicate, or, more specifically, what communication paradigm

is used?

• What (potentially changing) roles and responsibilities do they have in the overall

architecture?

• How are they mapped on to the physical distributed infrastructure (what is their

placement)?

Communicating entities • The first two questions above are absolutely central to an

understanding of distributed systems; what is communicating and how those entities

communicate together define a rich design space for the distributed systems developer

to consider. It is helpful to address the first question from a system-oriented and a

problem-oriented perspective.

From a system perspective, the answer is normally very clear in that the entities

that communicate in a distributed system are typically processes, leading to the

prevailing view of a distributed system as processes coupled with appropriate

interprocess communication paradigms (as discussed, for example, in Chapter 4), with

two caveats:

• In some primitive environments, such as sensor networks, the underlying

operating systems may not support process abstractions (or indeed any form of

isolation), and hence the entities that communicate in such systems are nodes.

• In most distributed system environments, processes are supplemented by threads,

so, strictly speaking, it is threads that are the endpoints of communication.

At one level, this is sufficient to model a distributed system and indeed the fundamental

models considered in Section 2.4 adopt this view. From a programming perspective,

however, this is not enough, and more problem-oriented abstractions have been

proposed:

Objects: Objects have been introduced to enable and encourage the use of objectoriented approaches in distributed systems (including both object-oriented design

and object-oriented programming languages). In distributed object-based

approaches, a computation consists of a number of interacting objects representing

natural units of decomposition for the given problem domain. Objects are accessed

via interfaces, with an associated interface definition language (or IDL) providing a

specification of the methods defined on an object. Distributed objects have become

a major area of study in distributed systems, and further consideration is given to this

topic in Chapters 5 and 8.

Components: Since their introduction a number of significant problems have been

identified with distributed objects, and the use of component technology has emerged

as a direct response to such weaknesses. Components resemble objects in that they

offer problem-oriented abstractions for building distributed systems and are also

accessed through interfaces. The key difference is that components specify not only

their (provided) interfaces but also the assumptions they make in terms of other

components/interfaces that must be present for a component to fulfil its function – in

other words, making all dependencies explicit and providing a more complete

contract for system construction. This more contractual approach encourages andSECTION 2.3 ARCHITECTURAL MODELS 59

enables third-party development of components and also promotes a purer

compositional approach to constructing distributed systems by removing hidden

dependencies. Component-based middleware often provides additional support for

key areas such as deployment and support for server-side programming [Heineman

and Councill 2001]. Further details of component-based approaches can be found in

Chapter 8.

Web services: Web services represent the third important paradigm for the

development of distributed systems [Alonso et al. 2004]. Web services are closely

related to objects and components, again taking an approach based on encapsulation

of behaviour and access through interfaces. In contrast, however, web services are

intrinsically integrated into the World Wide Web, using web standards to represent

and discover services. The World Wide Web consortium (W3C) defines a web

service as:

... a software application identified by a URI, whose interfaces and

bindings are capable of being defined, described and discovered as XML

artefacts. A Web service supports direct interactions with other software

agents using XML-based message exchanges via Internet-based

protocols.

In other words, web services are partially defined by the web-based technologies they

adopt. A further important distinction stems from the style of use of the technology.

Whereas objects and components are often used within an organization to develop

tightly coupled applications, web services are generally viewed as complete services

in their own right that can be combined to achieve value-added services, often

crossing organizational boundaries and hence achieving business to business

integration. Web services may be implemented by different providers and using

different underlying technologies. Web services are considered further in Chapter 9.

Communication paradigms • We now turn our attention to how entities communicate in

a distributed system, and consider three types of communication paradigm:

• interprocess communication;

• remote invocation;

• indirect communication.

Interprocess communication refers to the relatively low-level support for

communication between processes in distributed systems, including message-passing

primitives, direct access to the API offered by Internet protocols (socket programming)

and support for multicast communication. Such services are discussed in detail in

Chapter 4.

Remote invocation represents the most common communication paradigm in

distributed systems, covering a range of techniques based on a two-way exchange

between communicating entities in a distributed system and resulting in the calling of a

remote operation, procedure or method, as defined further below (and considered fully

in Chapter 5):

Request-reply protocols: Request-reply protocols are effectively a pattern imposed

on an underlying message-passing service to support client-server computing. In60 CHAPTER 2 SYSTEM MODELS

particular, such protocols typically involve a pairwise exchange of messages from

client to server and then from server back to client, with the first message containing

an encoding of the operation to be executed at the server and also an array of bytes

holding associated arguments and the second message containing any results of the

operation, again encoded as an array of bytes. This paradigm is rather primitive and

only really used in embedded systems where performance is paramount. The

approach is also used in the HTTP protocol described in Section 5.2. Most distributed

systems will elect to use remote procedure calls or remote method invocation, as

discussed below, but note that both approaches are supported by underlying requestreply exchanges.

Remote procedure calls: The concept of a remote procedure call (RPC), initially

attributed to Birrell and Nelson [1984], represents a major intellectual breakthrough

in distributed computing. In RPC, procedures in processes on remote computers can

be called as if they are procedures in the local address space. The underlying RPC

system then hides important aspects of distribution, including the encoding and

decoding of parameters and results, the passing of messages and the preserving of the

required semantics for the procedure call. This approach directly and elegantly

supports client-server computing with servers offering a set of operations through a

service interface and clients calling these operations directly as if they were available

locally. RPC systems therefore offer (at a minimum) access and location

transparency.

Remote method invocation: Remote method invocation (RMI) strongly resembles

remote procedure calls but in a world of distributed objects. With this approach, a

calling object can invoke a method in a remote object. As with RPC, the underlying

details are generally hidden from the user. RMI implementations may, though, go

further by supporting object identity and the associated ability to pass object

identifiers as parameters in remote calls. They also benefit more generally from

tighter integration into object-oriented languages as discussed in Chapter 5.

The above set of techniques all have one thing in common: communication represents a

two-way relationship between a sender and a receiver with senders explicitly directing

messages/invocations to the associated receivers. Receivers are also generally aware of

the identity of senders, and in most cases both parties must exist at the same time. In

contrast, a number of techniques have emerged whereby communication is indirect,

through a third entity, allowing a strong degree of decoupling between senders and

receivers. In particular:

• Senders do not need to know who they are sending to (space uncoupling).

• Senders and receivers do not need to exist at the same time (time uncoupling).

Indirect communication is discussed in more detail in Chapter 6.

Key techniques for indirect communication include:

Group communication: Group communication is concerned with the delivery of

messages to a set of recipients and hence is a multiparty communication paradigm

supporting one-to-many communication. Group communication relies on the

abstraction of a group which is represented in the system by a group identifier.SECTION 2.3 ARCHITECTURAL MODELS 61

Recipients elect to receive messages sent to a group by joining the group. Senders

then send messages to the group via the group identifier, and hence do not need to

know the recipients of the message. Groups typically also maintain group

membership and include mechanisms to deal with failure of group members.

Publish-subscribe systems: Many systems, such as the financial trading example in

Chapter 1, can be classified as information-dissemination systems wherein a large

number of producers (or publishers) distribute information items of interest (events)

to a similarly large number of consumers (or subscribers). It would be complicated

and inefficient to employ any of the core communication paradigms discussed above

for this purpose and hence publish-subscribe systems (sometimes also called

distributed event-based systems) have emerged to meet this important need [Muhl et

al. 2006]. Publish-subscribe systems all share the crucial feature of providing an

intermediary service that efficiently ensures information generated by producers is

routed to consumers who desire this information.

Message queues: Whereas publish-subscribe systems offer a one-to-many style of

communication, message queues offer a point-to-point service whereby producer

processes can send messages to a specified queue and consumer processes can

receive messages from the queue or be notified of the arrival of new messages in the

queue. Queues therefore offer an indirection between the producer and consumer

processes.

Tuple spaces: Tuple spaces offer a further indirect communication service by

supporting a model whereby processes can place arbitrary items of structured data,

called tuples, in a persistent tuple space and other processes can either read or remove

such tuples from the tuple space by specifying patterns of interest. Since the tuple

space is persistent, readers and writers do not need to exist at the same time. This style

of programming, otherwise known as generative communication, was introduced by

Gelernter [1985] as a paradigm for parallel programming. A number of distributed

implementations have also been developed, adopting either a client-server-style

implementation or a more decentralized peer-to-peer approach.

Distributed shared memory: Distributed shared memory (DSM) systems provide an

abstraction for sharing data between processes that do not share physical memory.

Programmers are nevertheless presented with a familiar abstraction of reading or

writing (shared) data structures as if they were in their own local address spaces, thus

presenting a high level of distribution transparency. The underlying infrastructure

must ensure a copy is provided in a timely manner and also deal with issues relating

to synchronization and consistency of data. An overview of distributed shared

memory can be found in Chapter 6.

The architectural choices discussed so far are summarized in Figure 2.2.

Roles and responsibilities • In a distributed system processes – or indeed objects,

components or services, including web services (but for the sake of simplicity we use

the term process throughout this section) – interact with each other to perform a useful

activity, for example, to support a chat session. In doing so, the processes take on given

roles, and these roles are fundamental in establishing the overall architecture to be62 CHAPTER 2 SYSTEM MODELS

adopted. In this section, we examine two architectural styles stemming from the role of

individual processes: client-server and peer-to-peer.

Client-server: This is the architecture that is most often cited when distributed systems

are discussed. It is historically the most important and remains the most widely

employed. Figure 2.3 illustrates the simple structure in which processes take on the roles

of being clients or servers. In particular, client processes interact with individual server

processes in potentially separate host computers in order to access the shared resources

that they manage.

Servers may in turn be clients of other servers, as the figure indicates. For

example, a web server is often a client of a local file server that manages the files in

which the web pages are stored. Web servers and most other Internet services are clients

of the DNS service, which translates Internet domain names to network addresses.

Another web-related example concerns search engines, which enable users to look up

summaries of information available on web pages at sites throughout the Internet. These

summaries are made by programs called web crawlers, which run in the background at

a search engine site using HTTP requests to access web servers throughout the Internet.

Thus a search engine is both a server and a client: it responds to queries from browser

clients and it runs web crawlers that act as clients of other web servers. In this example,

the server tasks (responding to user queries) and the crawler tasks (making requests to

other web servers) are entirely independent; there is little need to synchronize them and

they may run concurrently. In fact, a typical search engine would normally include many

concurrent threads of execution, some serving its clients and others running web

crawlers. In Exercise 2.5, the reader is invited to consider the only synchronization issue

that does arise for a concurrent search engine of the type outlined here.

Figure 2.2 Communicating entities and communication paradigms

Communicating entities

(what is communicating)

Communication paradigms

(how they communicate)

System-oriented

entities

Problemoriented entities

Interprocess

communication

Remote

invocation

Indirect

communication

Nodes

Processes

Objects

Components

Web services

Message

passing

Sockets

Multicast

Requestreply

RPC

RMI

Group

communication

Publish-subscribe

Message queues

Tuple spaces

DSMSECTION 2.3 ARCHITECTURAL MODELS 63

Peer-to-peer: In this architecture all of the processes involved in a task or activity play

similar roles, interacting cooperatively as peers without any distinction between client

and server processes or the computers on which they run. In practical terms, all

participating processes run the same program and offer the same set of interfaces to each

other. While the client-server model offers a direct and relatively simple approach to the

sharing of data and other resources, it scales poorly. The centralization of service

provision and management implied by placing a service at a single address does not

scale well beyond the capacity of the computer that hosts the service and the bandwidth

of its network connections.

A number of placement strategies have evolved in response to this problem (see

the discussion of placement below), but none of them addresses the fundamental issue

– the need to distribute shared resources much more widely in order to share the

computing and communication loads incurred in accessing them amongst a much larger

number of computers and network links. The key insight that led to the development of

peer-to-peer systems is that the network and computing resources owned by the users of

a service could also be put to use to support that service. This has the useful consequence

that the resources available to run the service grow with the number of users.

The hardware capacity and operating system functionality of today’s desktop

computers exceeds that of yesterday’s servers, and the majority are equipped with

always-on broadband network connections. The aim of the peer-to-peer architecture is

to exploit the resources (both data and hardware) in a large number of participating

computers for the fulfilment of a given task or activity. Peer-to-peer applications and

systems have been successfully constructed that enable tens or hundreds of thousands of

computers to provide access to data and other resources that they collectively store and

manage. One of the earliest instances was the Napster application for sharing digital

music files. Although Napster was not a pure peer-to-peer architecture (and also gained

notoriety for reasons beyond its architecture), its demonstration of feasibility has

resulted in the development of the architectural model in many valuable directions. A

more recent and widely used instance is the BitTorrent file-sharing system (discussed in

more depth in Section 20.6.2).

Figure 2.3 Clients invoke individual servers

invocation

result

invocation

result

Process:

Key:

Computer:

Client

Client

Server

Server64 CHAPTER 2 SYSTEM MODELS

Figure 2.4 illustrates the form of a peer-to-peer application. Applications are

composed of large numbers of peer processes running on separate computers and the

pattern of communication between them depends entirely on application requirements.

A large number of data objects are shared, an individual computer holds only a small

part of the application database, and the storage, processing and communication loads

for access to objects are distributed across many computers and network links. Each

object is replicated in several computers to further distribute the load and to provide

resilience in the event of disconnection of individual computers (as is inevitable in the

large, heterogeneous networks at which peer-to-peer systems are aimed). The need to

place individual objects and retrieve them and to maintain replicas amongst many

computers renders this architecture substantially more complex than the client-server

architecture.

The development of peer-to-peer applications and middleware to support them is

described in depth in Chapter 10.

Placement • The final issue to be considered is how entities such as objects or services

map on to the underlying physical distributed infrastructure which will consist of a

potentially large number of machines interconnected by a network of arbitrary

complexity. Placement is crucial in terms of determining the properties of the distributed

system, most obviously with regard to performance but also to other aspects, such as

reliability and security.

The question of where to place a given client or server in terms of machines and

processes within machines is a matter of careful design. Placement needs to take into

account the patterns of communication between entities, the reliability of given

machines and their current loading, the quality of communication between different

machines and so on. Placement must be determined with strong application knowledge,

and there are few universal guidelines to obtaining an optimal solution. We therefore

focus mainly on the following placement strategies, which can significantly alter the

characteristics of a given design (although we return to the key issue of mapping to

physical infrastructure in Section 2.3.2 , where we look at tiered architecture):

Figure 2.4 A service provided by multiple servers

Service

Client

Client

Server

Server

ServerSECTION 2.3 ARCHITECTURAL MODELS 65

• mapping of services to multiple servers;

• caching;

• mobile code;

• mobile agents.

Mapping of services to multiple servers: Services may be implemented as several server

processes in separate host computers interacting as necessary to provide a service to

client processes (Figure 2.4). The servers may partition the set of objects on which the

service is based and distribute those objects between themselves, or they may maintain

replicated copies of them on several hosts. These two options are illustrated by the

following examples.

The Web provides a common example of partitioned data in which each web

server manages its own set of resources. A user can employ a browser to access a

resource at any one of the servers.

An example of a service based on replicated data is the Sun Network Information

Service (NIS), which is used to enable all the computers on a LAN to access the same

user authentication data when users log in. Each NIS server has its own replica of a

common password file containing a list of users’ login names and encrypted passwords.

Chapter 18 discusses techniques for replication in detail.

A more closely coupled type of multiple-server architecture is the cluster, as

introduced in Chapter 1. A cluster is constructed from up to thousands of commodity

processing boards, and service processing can be partitioned or replicated between

them.

Caching: A cache is a store of recently used data objects that is closer to one client or a

particular set of clients than the objects themselves. When a new object is received from

a server it is added to the local cache store, replacing some existing objects if necessary.

When an object is needed by a client process, the caching service first checks the cache

and supplies the object from there if an up-to-date copy is available. If not, an up-to-date

copy is fetched. Caches may be co-located with each client or they may be located in a

proxy server that can be shared by several clients.

Caches are used extensively in practice. Web browsers maintain a cache of

recently visited web pages and other web resources in the client’s local file system, using

a special HTTP request to check with the original server that cached pages are up-todate before displaying them. Web proxy servers (Figure 2.5) provide a shared cache of

Client

Proxy

Web

Figure 2.5 Web proxy server

server

Web

server

server

Client66 CHAPTER 2 SYSTEM MODELS

web resources for the client machines at a site or across several sites. The purpose of

proxy servers is to increase the availability and performance of the service by reducing

the load on the wide area network and web servers. Proxy servers can take on other roles;

for example, they may be used to access remote web servers through a firewall.

Mobile code: Chapter 1 introduced mobile code. Applets are a well-known and widely

used example of mobile code – the user running a browser selects a link to an applet

whose code is stored on a web server; the code is downloaded to the browser and runs

there, as shown in Figure 2.6. An advantage of running the downloaded code locally is

that it can give good interactive response since it does not suffer from the delays or

variability of bandwidth associated with network communication.

Accessing services means running code that can invoke their operations. Some

services are likely to be so standardized that we can access them with an existing and

well-known application – the Web is the most common example of this, but even there,

some web sites use functionality not found in standard browsers and require the

downloading of additional code. The additional code may, for example, communicate

with the server. Consider an application that requires that users be kept up-to-date with

changes as they occur at an information source in the server. This cannot be achieved by

normal interactions with the web server, which are always initiated by the client. The

solution is to use additional software that operates in a manner often referred to as a push

model – one in which the server instead of the client initiates interactions. For example,

a stockbroker might provide a customized service to notify customers of changes in the

prices of shares; to use the service, each customer would have to download a special

applet that receives updates from the broker’s server, displays them to the user and

perhaps performs automatic buy and sell operations triggered by conditions set up by the

customer and stored locally in the customer’s computer.

Mobile code is a potential security threat to the local resources in the destination

computer. Therefore browsers give applets limited access to local resources, using a

scheme discussed in Section 11.1.1.

Mobile agents: A mobile agent is a running program (including both code and data) that

travels from one computer to another in a network carrying out a task on someone’s

behalf, such as collecting information, and eventually returning with the results. A

mobile agent may make many invocations to local resources at each site it visits – for

a) client request results in the downloading of applet code

Web

Figure 2.6 Web applets

server

Client

Web

Applet server

Applet code

Client

b) client interacts with the appletSECTION 2.3 ARCHITECTURAL MODELS 67

example, accessing individual database entries. If we compare this architecture with a

static client making remote invocations to some resources, possibly transferring large

amounts of data, there is a reduction in communication cost and time through the

replacement of remote invocations with local ones.

Mobile agents might be used to install and maintain software on the computers

within an organization or to compare the prices of products from a number of vendors

by visiting each vendor’s site and performing a series of database operations. An early

example of a similar idea is the so-called worm program developed at Xerox PARC

[Shoch and Hupp 1982], which was designed to make use of idle computers in order to

carry out intensive computations.

Mobile agents (like mobile code) are a potential security threat to the resources in

computers that they visit. The environment receiving a mobile agent should decide

which of the local resources it should be allowed to use, based on the identity of the user

on whose behalf the agent is acting – their identity must be included in a secure way with

the code and data of the mobile agent. In addition, mobile agents can themselves be

vulnerable – they may not be able to complete their task if they are refused access to the

information they need. The tasks performed by mobile agents can be performed by other

means. For example, web crawlers that need to access resources at web servers

throughout the Internet work quite successfully by making remote invocations to server

processes. For these reasons, the applicability of mobile agents may be limited.

2.3.2 Architectural patterns

Architectural patterns build on the more primitive architectural elements discussed

above and provide composite recurring structures that have been shown to work well in

given circumstances. They are not themselves necessarily complete solutions but rather

offer partial insights that, when combined with other patterns, lead the designer to a

solution for a given problem domain.

This is a large topic, and many architectural patterns have been identified for

distributed systems. In this section, we present several key architectural patterns in

distributed systems, including layering and tiered architectures and the related concept

of thin clients (including the specific mechanism of virtual network computing). We

also examine web services as an architectural pattern and give pointers to others that

may be applicable in distributed systems.

Layering • The concept of layering is a familiar one and is closely related to abstraction.

In a layered approach, a complex system is partitioned into a number of layers, with a

given layer making use of the services offered by the layer below. A given layer

therefore offers a software abstraction, with higher layers being unaware of

implementation details, or indeed of any other layers beneath them.

In terms of distributed systems, this equates to a vertical organization of services

into service layers. A distributed service can be provided by one or more server

processes, interacting with each other and with client processes in order to maintain a

consistent system-wide view of the service’s resources. For example, a network time

service is implemented on the Internet based on the Network Time Protocol (NTP) by

server processes running on hosts throughout the Internet that supply the current time to

any client that requests it and adjust their version of the current time as a result of68 CHAPTER 2 SYSTEM MODELS

interactions with each other. Given the complexity of distributed systems, it is often

helpful to organize such services into layers. We present a common view of a layered

architecture in Figure 2.7 and develop this view in increasing detail in Chapters 3 to 6.

Figure 2.7 introduces the important terms platform and middleware, which we

define as follows:

• A platform for distributed systems and applications consists of the lowest-level

hardware and software layers. These low-level layers provide services to the

layers above them, which are implemented independently in each computer,

bringing the system’s programming interface up to a level that facilitates

communication and coordination between processes. Intel x86/Windows, Intel

x86/Solaris, Intel x86/Mac OS X, Intel x86/Linux and ARM/Symbian are major

examples.

• Middleware was defined in Section 1.5.1 as a layer of software whose purpose is

to mask heterogeneity and to provide a convenient programming model to

application programmers. Middleware is represented by processes or objects in a

set of computers that interact with each other to implement communication and

resource-sharing support for distributed applications. It is concerned with

providing useful building blocks for the construction of software components that

can work with one another in a distributed system. In particular, it raises the level

of the communication activities of application programs through the support of

abstractions such as remote method invocation; communication between a group

of processes; notification of events; the partitioning, placement and retrieval of

shared data objects amongst cooperating computers; the replication of shared data

objects; and the transmission of multimedia data in real time. We return to this

important topic in Section 2.3.3 below.

Tiered architecture • Tiered architectures are complementary to layering. Whereas

layering deals with the vertical organization of services into layers of abstraction, tiering

is a technique to organize functionality of a given layer and place this functionality into

Figure 2.7 Software and hardware service layers in distributed systems

Applications, services

Computer and network hardware

Platform

Operating system

MiddlewareSECTION 2.3 ARCHITECTURAL MODELS 69

appropriate servers and, as a secondary consideration, on to physical nodes. This

technique is most commonly associated with the organization of applications and

services as in Figure 2.7 above, but it also applies to all layers of a distributed systems

architecture.

Let us first examine the concepts of two- and three-tiered architecture. To

illustrate this, consider the functional decomposition of a given application, as follows:

• the presentation logic, which is concerned with handling user interaction and

updating the view of the application as presented to the user;

• the application logic, which is concerned with the detailed application-specific

processing associated with the application (also referred to as the business logic,

although the concept is not limited only to business applications);

• the data logic, which is concerned with the persistent storage of the application,

typically in a database management system.

Now, let us consider the implementation of such an application using client-server

technology. The associated two-tier and three-tier solutions are presented together for

comparison in Figure 2.8 (a) and (b), respectively.

In the two-tier solution, the three aspects mentioned above must be partitioned

into two processes, the client and the server. This is most commonly done by splitting

the application logic, with some residing in the client and the remainder in the server

(although other solutions are also possible). The advantage of this scheme is low latency

in terms of interaction, with only one exchange of messages to invoke an operation. The

disadvantage is the splitting of application logic across a process boundary, with the

consequent restriction on which parts of the logic can be directly invoked from which

other part.

In the three-tier solution, there is a one-to-one mapping from logical elements to

physical servers and hence, for example, the application logic is held in one place, which

in turn can enhance maintainability of the software. Each tier also has a well-defined

role; for example, the third tier is simply a database offering a (potentially standardized)

relational service interface. The first tier can also be a simple user interface allowing

intrinsic support for thin clients (as discussed below). The drawbacks are the added

complexity of managing three servers and also the added network traffic and latency

associated with each operation.

Note that this approach generalizes to n-tiered (or multi-tier) solutions where a

given application domain is partitioned into n logical elements, each mapped to a given

server element. As an example, Wikipedia, the web-based publicly editable

encyclopedia, adopts a multi-tier architecture to deal with the high volume of web

requests (up to 60,000 page requests per second).

The role of AJAX: In Section 1.6 we introduced AJAX (Asynchronous Javascript And

XML) as an extension to the standard client-server style of interaction used in the World

Wide Web. AJAX meets the need for fine-grained communication between a Javascript

front-end program running in a web browser and a server-based back-end program

holding data describing the state of the application. To recapitulate, in the standard web

style of interaction a browser sends an HTTP request to a server for a page, image or

other resource with a given URL. The server replies by sending an entire page that is

either read from a file on the server or generated by a program, depending on which type70 CHAPTER 2 SYSTEM MODELS

of resource is identified in the URL. When the resultant content is received at the client,

the browser presents it according to the relevant display method for its MIME type

(text/html, image/jpg, etc.). Although a web page may be composed of several items of

content of different types, the entire page is composed and presented by the browser in

the manner specified in its HTML page definition.

This standard style of interaction constrains the development of web applications

in several significant ways:

• Once the browser has issued an HTTP request for a new web page, the user is

unable to interact with the page until the new HTML content is received and

presented by the browser. This time interval is indeterminate, because it is subject

to network and server delays.

• In order to update even a small part of the current page with additional data from

the server, an entire new page must be requested and displayed. This results in a

delayed response to the user, additional processing at both the client and the server

and redundant network traffic.

Figure 2.8 Two-tier and three-tier architectures

Application

and data management

Personal computers

Server

b)

User view,

controls and

or mobile devices

Tier 1 Tier 2

User

view and Application

logic

Personal computers Application server

controls

User

view and

controls

or mobile devices

Database server

Application

logic

Database

manager

Tier 1 Tier 2 Tier 3

a)

data manipulation

Application

and data management

User view,

controls and

data manipulationSECTION 2.3 ARCHITECTURAL MODELS 71

• The contents of a page displayed at a client cannot be updated in response to

changes in the application data held at the server.

The introduction of Javascript, a cross-platform and cross-browser programming

language that is downloaded and executed in the browser, constituted a first step towards

the removal of those constraints. Javascript is a general-purpose language enabling both

user interface and application logic to be programmed and executed in the context of a

browser window.

AJAX is the second innovative step that was needed to enable major interactive

web applications to be developed and deployed. It enables Javascript front-end

programs to request new data directly from server programs. Any data items can be

requested and the current page updated selectively to show the new values. Indeed, the

front end can react to the new data in any way that is useful for the application.

Many web applications allow users to access and update substantial shared

datasets that may be subject to change in response to input from other clients or data

feeds received by a server. They require a responsive front-end component running in

each client browser to perform user interface actions such as menu selection, but they

also require access to a dataset that must be held at server to enable sharing. Such

datasets are generally too large and too dynamic to allow the use of any architecture

based on the downloading of a copy of the entire application state to the client at the start

of a user’s session for manipulation by the client.

AJAX is the ‘glue’ that supports the construction of such applications; it provides

a communication mechanism enabling front-end components running in a browser to

issue requests and receive results from back-end components running on a server.

Clients issue requests through the Javascript XmlHttpRequest object, which manages an

HTTP exchange (see Section 1.6) with a server process. Because XmlHttpRequest has a

complex API that is also somewhat browser-dependent, it is usually accessed through

one of the many Javascript libraries that are available to support the development of web

applications. In Figure 2.9 we illustrate its use in the Prototype.js Javascript library

[www.prototypejs.org].

The example is an excerpt from a web application that displays a page listing upto-date scores for soccer matches. Users may request updates of scores for individual

games by clicking on the relevant line of the page, which executes the first line of the

Figure 2.9 AJAX example: soccer score updates

new Ajax.Request('scores.php?game=Arsenal:Liverpool’,

{onSuccess: updateScore});

function updateScore(request) {

.....

( request contains the state of the Ajax request including the returned result.

The result is parsed to obtain some text giving the score, which is used

to update the relevant portion of the current page.)

.....

}72 CHAPTER 2 SYSTEM MODELS

example. The Ajax.Request object sends an HTTP request to a scores.php program

located at the same server as the web page. The Ajax.Request object then returns control,

allowing the browser to continue to respond to other user actions in the same window or

other windows. When the scores.php program has obtained the latest score it returns it

in an HTTP response. The Ajax.Request object is then reactivated; it invokes the

updateScore function (because it is the onSuccess action), which parses the result and

inserts the score at the relevant position in the current page. The remainder of the page

remains unaffected and is not reloaded.

This illustrates the type of communication used between Tier 1 and Tier 2

components. Although Ajax.Request (and the underlying XmlHttpRequest object) offers

both synchronous and asynchronous communication, the asynchronous version is

almost always used because the effect on the user interface of delayed server responses

is unacceptable.

Our simple example illustrates the use of AJAX in a two-tier application. In a

three-tier application the server component (scores.php in our example) would send a

request to a data manager component (typically an SQL query to a database server) for

the required data. That request would be synchronous, since there is no reason to return

control to the server component until the request is satisfied.

The AJAX mechanism constitutes an effective technique for the construction of

responsive web applications in the context of the indeterminate latency of the Internet,

and it has been very widely deployed. The Google Maps application [www.google.com

II] is an outstanding example. Maps are displayed as an array of contiguous 256 x 256

pixel images (called tiles). When the map is moved the visible tiles are repositioned by

Javascript code in the browser and additional tiles needed to fill the visible area are

requested with an AJAX call to a Google server. They are displayed as soon as they are

received, but the browser continues to respond to user interaction while they are awaited.

Thin clients • The trend in distributed computing is towards moving complexity away

from the end-user device towards services in the Internet. This is most apparent in the

move towards cloud computing (discussed in Chapter 1) but can also be seen in tiered

architectures, as discussed above. This trend has given rise to interest in the concept of

a thin client, enabling access to sophisticated networked services, provided for example

by a cloud solution, with few assumptions or demands on the client device. More

specifically, the term thin client refers to a software layer that supports a window-based

user interface that is local to the user while executing application programs or, more

generally, accessing services on a remote computer. For example, Figure 2.10 illustrates

a thin client accessing a compute server over the Internet. The advantage of this

approach is that potentially simple local devices (including, for example, smart phones

Figure 2.10 Thin clients and computer servers

Thin

Client

Application

Process

Networked device Compute server

NetworkSECTION 2.3 ARCHITECTURAL MODELS 73

and other resource-constrained devices) can be significantly enhanced with a plethora of

networked services and capabilities. The main drawback of the thin client architecture

is in highly interactive graphical activities such as CAD and image processing, where

the delays experienced by users are increased to unacceptable levels by the need to

transfer image and vector information between the thin client and the application

process, due to both network and operating system latencies.

This concept has led to the emergence of virtual network computing (VNC). This

technology was first introduced by researchers at the Olivetti and Oracle Research

Laboratory [Richardson et al. 1998]; the initial concept has now evolved into

implementations such as RealVNC [www.realvnc.com], which is a software solution,

and Adventiq [www.adventiq.com], which is a hardware-based solution supporting the

transmission of keyboard, video and mouse events over IP (KVM-over-IP). Other VNC

implementationss include Apple Remote Desktop, TightVNC and Aqua Connect.

The concept is straightforward, providing remote access to graphical user

interfaces. In this solution, a VNC client (or viewer) interacts with a VNC server through

a VNC protocol. The protocol operates at a primitive level in terms of graphics support,

based on framebuffers and featuring one operation: the placement of a rectangle of pixel

data at a given position on the screen (some solutions, such as XenApp from Citrix

operate at a higher level in terms of window operations [www.citrix.com]). This lowlevel approach ensures the protocol will work with any operating system or application.

Although it is straightforward, the implication is that users are able to access their

computer facilities from anywhere on a wide range of devices, representing a significant

step forward in mobile computing.

Virtual network computing has superseded network computers, a previous

attempt to realise thin client solutions through simple and inexpensive hardware devices

that are completely reliant on networked services, downloading their operating system

and any application software needed by the user from a remote file server. Since all the

application data and code is stored by a file server, the users may migrate from one

network computer to another. In practice, virtual network computing has proved to be a

more flexible solution and now dominates the marketplace.

Other commonly occurring patterns • As mentioned above, a large number of

architectural patterns have now been identified and documented. Here are a few key

examples:

• The proxy pattern is a commonly recurring pattern in distributed systems designed

particularly to support location transparency in remote procedure calls or remote

method invocation. With this approach, a proxy is created in the local address

space to represent the remote object. This proxy offers exactly the same interface

as the remote object, and the programmer makes calls on this proxy object and

hence does not need to be aware of the distributed nature of the interaction. The

role of proxies in supporting such location transparency in RPC and RMI is

discussed further in Chapter 5. Note that proxies can also be used to encapsulate

other functionality, such as the placement policies of replication or caching.

• The use of brokerage in web services can usefully be viewed as an architectural

pattern supporting interoperability in potentially complex distributed

infrastructures. In particular, this pattern consists of the trio of service provider,74 CHAPTER 2 SYSTEM MODELS

service requester and service broker (a service that matches services provided to

those requested), as shown in Figure 2.11. This brokerage pattern is replicated in

many areas of distributed systems, for example with the registry in Java RMI and

the naming service in CORBA (as discussed in Chapters 5 and 8, respectively).

• Reflection is a pattern that is increasingly being used in distributed systems as a

means of supporting both introspection (the dynamic discovery of properties of

the system) and intercession (the ability to dynamically modify structure or

behaviour). For example, the introspection capabilities of Java are used

effectively in the implementation of RMI to provide generic dispatching (as

discussed in Section 5.4.2). In a reflective system, standard service interfaces are

available at the base level, but a meta-level interface is also available providing

access to the components and their parameters involved in the realization of the

services. A variety of techniques are generally available at the meta-level,

including the ability to intercept incoming messages or invocations, to

dynamically discover the interface offered by a given object and to discover and

adapt the underlying architecture of the system. Reflection has been applied in a

variety of areas in distributed systems, particularly within the field of reflective

middleware, for example to support more configurable and reconfigurable

middleware architectures [Kon et al. 2002].

Further examples of architectural patterns related to distributed systems can be found in

Bushmann et al. [2007].

2.3.3 Associated middleware solutions

Middleware has already been introduced in Chapter 1 and revisited in the discussion of

layering in Section 2.3.2 above. The task of middleware is to provide a higher-level

programming abstraction for the development of distributed systems and, through

layering, to abstract over heterogeneity in the underlying infrastructure to promote

interoperability and portability. Middleware solutions are based on the architectural

models introduced in Section 2.3.1 and also support more complex architectural

Figure 2.11 The web service architectural pattern

Service

Provider

Service

Broker

Service

RequesterSECTION 2.3 ARCHITECTURAL MODELS 75

patterns. In this section, we briefly review the major classes of middleware that exist

today and prepare the ground for further study of these solutions in the rest of the book.

Categories of middleware • Remote procedure calling packages such as Sun RPC

(Chapter 5) and group communication systems such as ISIS (Chapters 6 and 18) were

amongst the earliest instances of middleware. Since then a wide range of styles of

middleware have emerged, based largely on the architectural models introduced above.

We present a taxonomy of such middleware platforms in Figure 2.12, including crossreferences to other chapters that cover the various categories in more detail. It must be

stressed that the categorizations are not exact and that modern middleware platforms

tend to offer hybrid solutions. For example, many distributed object platforms offer

distributed event services to complement the more traditional support for remote method

invocation. Similarly, many component-based platforms (and indeed other categories of

platform) also support web service interfaces and standards, for reasons of

interoperability. It should also be stressed that this taxonomy is not intended to be

complete in terms of the set of middleware standards and technologies available today,

Figure 2.12 Categories of middleware

Major categories: Subcategory Example systems

Distributed objects (Chapters 5, 8) Standard RM-ODP

Platform CORBA

Platform Java RMI

Distributed components (Chapter 8) Lightweight components Fractal

Lightweight components OpenCOM

Application servers SUN EJB

Application servers CORBA Component Model

Application servers JBoss

Publish-subscribe systems (Chapter 6) - CORBA Event Service

- Scribe

- JMS

Message queues (Chapter 6) - Websphere MQ

- JMS

Web services (Chapter 9) Web services Apache Axis

Grid services The Globus Toolkit

Peer-to-peer (Chapter 10) Routing overlays Pastry

Routing overlays Tapestry

Application-specific Squirrel

Application-specific OceanStore

Application-specific Ivy

Application-specific Gnutella76 CHAPTER 2 SYSTEM MODELS

but rather is intended to be indicative of the major classes of middleware. Other

solutions (not shown) tend to be more specific, for example offering particular

communication paradigms such as message passing, remote procedure calls, distributed

shared memory, tuple spaces or group communication.

The top-level categorization of middleware in Figure 2.12 is driven by the choice

of communicating entities and associated communication paradigms, and follows five

of the main architectural models: distributed objects, distributed components, publishsubscribe systems, message queues and web services. These are supplemented by peerto-peer systems, a rather separate branch of middleware based on the cooperative

approach discussed in Section 2.3.1. The subcategory of distributed components shown

as application servers also provides direct support for three-tier architectures. In

particular, application servers provide structure to support a separation between

application logic and data storage, along with support for other properties such as

security and reliability. Further detail is deferred until Chapter 8.

In addition to programming abstractions, middleware can also provide

infrastructural distributed system services for use by application programs or other

services. These infrastructural services are tightly bound to the distributed programming

model that the middleware provides. For example, CORBA (Chapter 8) provides

applications with a range of CORBA services, including support for making

applications secure and reliable. As mentioned above and discussed further in Chapter

8, application servers also provide intrinsic support for such services.

Limitations of middleware • Many distributed applications rely entirely on the services

provided by middleware to support their needs for communication and data sharing. For

example, an application that is suited to the client-server model such as a database of

names and addresses, can rely on middleware that provides only remote method

invocation.

Much has been achieved in simplifying the programming of distributed systems

through the development of middleware support, but some aspects of the dependability

of systems require support at the application level.

Consider the transfer of large electronic mail messages from the mail host of the

sender to that of the recipient. At first sight this a simple application of the TCP data

transmission protocol (discussed in Chapter 3). But consider the problem of a user who

attempts to transfer a very large file over a potentially unreliable network. TCP provides

some error detection and correction, but it cannot recover from major network

interruptions. Therefore the mail transfer service adds another level of fault tolerance,

maintaining a record of progress and resuming transmission using a new TCP

connection if the original one breaks.

A classic paper by Saltzer, Reed and Clarke [Saltzer et al. 1984] makes a similar

and valuable point about the design of distributed systems, which they call the ‘the endto-end argument’. To paraphrase their statement:

Some communication-related functions can be completely and reliably

implemented only with the knowledge and help of the application standing at the

end points of the communication system. Therefore, providing that function as a

feature of the communication system itself is not always sensible. (Although an

incomplete version of the function provided by the communication system may

sometimes be useful as a performance enhancement).SECTION 2.4 FUNDAMENTAL MODELS 77

It can be seen that their argument runs counter to the view that all communication

activities can be abstracted away from the programming of applications by the

introduction of appropriate middleware layers.

The nub of their argument is that correct behaviour in distributed programs

depends upon checks, error-correction mechanisms and security measures at many

levels, some of which require access to data within the application’s address space. Any

attempt to perform the checks within the communication system alone will guarantee

only part of the required correctness. The same work is therefore likely to be duplicated

in application programs, wasting programming effort and, more importantly, adding

unnecessary complexity and redundant computations.

There is not space to detail their arguments further here, but reading the cited

paper is strongly recommended – it is replete with illuminating examples. One of the

original authors has recently pointed out that the substantial benefits that the use of the

argument brought to the design of the Internet are placed at risk by recent moves towards

the specialization of network services to meet current application requirements

[www.reed.com].

This argument poses a real dilemma for middleware designers, and indeed the

difficulties are increasing given the wide range of applications (and associated

environmental conditions) in contemporary distributed systems (see Chapter 1). In

essence, the right underlying middleware behaviour is a function of the requirements of

a given application or set of applications and the associated environmental context, such

as the state and style of the underlying network. This perception is driving interest in

context-aware and adaptive solutions to middleware, as discussed in Kon et al [2002].

2.4 Fundamental models

All the above, quite different, models of systems share some fundamental properties. In

particular, all of them are composed of processes that communicate with one another by

sending messages over a computer network. All of the models share the design

requirements of achieving the performance and reliability characteristics of processes

and networks and ensuring the security of the resources in the system. In this section, we

present models based on the fundamental properties that allow us to be more specific

about their characteristics and the failures and security risks they might exhibit.

In general, such a fundamental model should contain only the essential ingredients

that we need to consider in order to understand and reason about some aspects of a

system’s behaviour. The purpose of such a model is:

• To make explicit all the relevant assumptions about the systems we are modelling.

• To make generalizations concerning what is possible or impossible, given those

assumptions. The generalizations may take the form of general-purpose

algorithms or desirable properties that are guaranteed. The guarantees are

dependent on logical analysis and, where appropriate, mathematical proof.

There is much to be gained by knowing what our designs do, and do not, depend upon.

It allows us to decide whether a design will work if we try to implement it in a particular

system: we need only ask whether our assumptions hold in that system. Also, by making78 CHAPTER 2 SYSTEM MODELS

our assumptions clear and explicit, we can hope to prove system properties using mathematical techniques. These properties will then hold for any system meeting our assumptions. Finally, by abstracting only the essential system entities and characteristics

away from details such as hardware, we can clarify our understanding of our systems.

The aspects of distributed systems that we wish to capture in our fundamental

models are intended to help us to discuss and reason about:

Interaction: Computation occurs within processes; the processes interact by passing

messages, resulting in communication (information flow) and coordination

(synchronization and ordering of activities) between processes. In the analysis and

design of distributed systems we are concerned especially with these interactions.

The interaction model must reflect the facts that communication takes place with

delays that are often of considerable duration, and that the accuracy with which

independent processes can be coordinated is limited by these delays and by the

difficulty of maintaining the same notion of time across all the computers in a

distributed system.

Failure: The correct operation of a distributed system is threatened whenever a fault

occurs in any of the computers on which it runs (including software faults) or in the

network that connects them. Our model defines and classifies the faults. This

provides a basis for the analysis of their potential effects and for the design of systems

that are able to tolerate faults of each type while continuing to run correctly.

Security: The modular nature of distributed systems and their openness exposes

them to attack by both external and internal agents. Our security model defines and

classifies the forms that such attacks may take, providing a basis for the analysis of

threats to a system and for the design of systems that are able to resist them.

As aids to discussion and reasoning, the models introduced in this chapter are

necessarily simplified, omitting much of the detail of real-world systems. Their

relationship to real-world systems, and the solution in that context of the problems that

the models help to bring out, is the main subject of this book.

2.4.1 Interaction model

The discussion of system architectures in Section 2.3 indicates that fundamentally

distributed systems are composed of many processes, interacting in complex ways. For

example:

• Multiple server processes may cooperate with one another to provide a service; the

examples mentioned above were the Domain Name System, which partitions and

replicates its data at servers throughout the Internet, and Sun’s Network

Information Service, which keeps replicated copies of password files at several

servers in a local area network.

• A set of peer processes may cooperate with one another to achieve a common

goal: for example, a voice conferencing system that distributes streams of audio

data in a similar manner, but with strict real-time constraints.

Most programmers will be familiar with the concept of an algorithm – a sequence of

steps to be taken in order to perform a desired computation. Simple programs areSECTION 2.4 FUNDAMENTAL MODELS 79

controlled by algorithms in which the steps are strictly sequential. The behaviour of the

program and the state of the program’s variables is determined by them. Such a program

is executed as a single process. Distributed systems composed of multiple processes

such as those outlined above are more complex. Their behaviour and state can be

described by a distributed algorithm – a definition of the steps to be taken by each of the

processes of which the system is composed, including the transmission of messages

between them. Messages are transmitted between processes to transfer information

between them and to coordinate their activity.

The rate at which each process proceeds and the timing of the transmission of

messages between them cannot in general be predicted. It is also difficult to describe all

the states of a distributed algorithm, because it must deal with the failures of one or more

of the processes involved or the failure of message transmissions.

Interacting processes perform all of the activity in a distributed system. Each

process has its own state, consisting of the set of data that it can access and update,

including the variables in its program. The state belonging to each process is completely

private – that is, it cannot be accessed or updated by any other process.

In this section, we discuss two significant factors affecting interacting processes

in a distributed system:

• Communication performance is often a limiting characteristic.

• It is impossible to maintain a single global notion of time.

Performance of communication channels • The communication channels in our model

are realized in a variety of ways in distributed systems – for example, by an

implementation of streams or by simple message passing over a computer network.

Communication over a computer network has the following performance characteristics

relating to latency, bandwidth and jitter:

• The delay between the start of a message’s transmission from one process and the

beginning of its receipt by another is referred to as latency. The latency includes:

– The time taken for the first of a string of bits transmitted through a network to

reach its destination. For example, the latency for the transmission of a

message through a satellite link is the time for a radio signal to travel to the

satellite and back.

– The delay in accessing the network, which increases significantly when the

network is heavily loaded. For example, for Ethernet transmission the sending

station waits for the network to be free of traffic.

– The time taken by the operating system communication services at both the

sending and the receiving processes, which varies according to the current load

on the operating systems.

• The bandwidth of a computer network is the total amount of information that can

be transmitted over it in a given time. When a large number of communication

channels are using the same network, they have to share the available bandwidth.

• Jitter is the variation in the time taken to deliver a series of messages. Jitter is

relevant to multimedia data. For example, if consecutive samples of audio data are

played with differing time intervals, the sound will be badly distorted.80 CHAPTER 2 SYSTEM MODELS

Computer clocks and timing events • Each computer in a distributed system has its own

internal clock, which can be used by local processes to obtain the value of the current

time. Therefore two processes running on different computers can each associate

timestamps with their events. However, even if the two processes read their clocks at the

same time, their local clocks may supply different time values. This is because computer

clocks drift from perfect time and, more importantly, their drift rates differ from one

another. The term clock drift rate refers to the rate at which a computer clock deviates

from a perfect reference clock. Even if the clocks on all the computers in a distributed

system are set to the same time initially, their clocks will eventually vary quite

significantly unless corrections are applied.

There are several approaches to correcting the times on computer clocks. For

example, computers may use radio receivers to get time readings from the Global

Positioning System with an accuracy of about 1 microsecond. But GPS receivers do not

operate inside buildings, nor can the cost be justified for every computer. Instead, a

computer that has an accurate time source such as GPS can send timing messages to

other computers in its network. The resulting agreement between the times on the local

clocks is, of course, affected by variable message delays. For a more detailed discussion

of clock drift and clock synchronization, see Chapter 14.

Two variants of the interaction model • In a distributed system it is hard to set limits on

the time that can be taken for process execution, message delivery or clock drift. Two

opposing extreme positions provide a pair of simple models – the first has a strong

assumption of time and the second makes no assumptions about time:

Synchronous distributed systems: Hadzilacos and Toueg [1994] define a

synchronous distributed system to be one in which the following bounds are defined:

• The time to execute each step of a process has known lower and upper bounds.

• Each message transmitted over a channel is received within a known bounded

time.

• Each process has a local clock whose drift rate from real time has a known

bound.

It is possible to suggest likely upper and lower bounds for process execution time,

message delay and clock drift rates in a distributed system, but it is difficult to arrive

at realistic values and to provide guarantees of the chosen values. Unless the values

of the bounds can be guaranteed, any design based on the chosen values will not be

reliable. However, modelling an algorithm as a synchronous system may be useful

for giving some idea of how it will behave in a real distributed system. In a

synchronous system it is possible to use timeouts, for example, to detect the failure

of a process, as shown in Section 2.4.2 below.

Synchronous distributed systems can be built. What is required is for the

processes to perform tasks with known resource requirements for which they can be

guaranteed sufficient processor cycles and network capacity, and for processes to be

supplied with clocks with bounded drift rates.SECTION 2.4 FUNDAMENTAL MODELS 81

Asynchronous distributed systems: Many distributed systems, such as the Internet,

are very useful without being able to qualify as synchronous systems. Therefore we

need an alternative model. An asynchronous distributed system is one in which there

are no bounds on:

• Process execution speeds – for example, one process step may take only a

picosecond and another a century; all that can be said is that each step may take

an arbitrarily long time.

• Message transmission delays – for example, one message from process A to

process B may be delivered in negligible time and another may take several

years. In other words, a message may be received after an arbitrarily long time.

• Clock drift rates – again, the drift rate of a clock is arbitrary.

The asynchronous model allows no assumptions about the time intervals involved in

any execution. This exactly models the Internet, in which there is no intrinsic bound

on server or network load and therefore on how long it takes, for example, to transfer

a file using FTP. Sometimes an email message can take days to arrive. The box on

this page illustrates the difficulty of reaching an agreement in an asynchronous

distributed system.

But some design problems can be solved even with these assumptions. For

example, although the Web cannot always provide a particular response within a

reasonable time limit, browsers have been designed to allow users to do other things

while they are waiting. Any solution that is valid for an asynchronous distributed

system is also valid for a synchronous one.

Actual distributed systems are very often asynchronous because of the need for

processes to share the processors and for communication channels to share the

Agreement in Pepperland • Two divisions of the Pepperland army, ‘Apple’ and

‘Orange’, are encamped at the top of two nearby hills. Further along the valley below

are the invading Blue Meanies. The Pepperland divisions are safe as long as they

remain in their encampments, and they can send out messengers reliably through the

valley to communicate. The Pepperland divisions need to agree on which of them

will lead the charge against the Blue Meanies and when the charge will take place.

Even in an asynchronous Pepperland, it is possible to agree on who will lead the

charge. For example, each division can send the number of its remaining members,

and the one with most will lead (if a tie, division Apple wins over Orange). But when

should they charge? Unfortunately, in asynchronous Pepperland, the messengers are

very variable in their speed. If, say, Apple sends a messenger with the message

‘Charge!’, Orange might not receive the message for, say, three hours; or it may take,

say, five minutes to arrive. In a synchronous Pepperland, there is still a coordination

problem, but the divisions know some useful constraints: every message takes at least

min minutes and at most max minutes to arrive. If the division that will lead the

charge sends a message ‘Charge!’, it waits for min minutes; then it charges. The other

division waits for 1 minute after receipt of the message, then charges. Its charge is

guaranteed to be after the leading division’s, but no more than (max – min + 1)

minutes after it.82 CHAPTER 2 SYSTEM MODELS

network. For example, if too many processes of unknown character are sharing a

processor, then the resulting performance of any one of them cannot be guaranteed.

But there are many design problems that cannot be solved for an asynchronous

system that can be solved when some aspects of time are used. The need for each

element of a multimedia data stream to be delivered before a deadline is such a

problem. For problems such as these, a synchronous model is required.

Event ordering • In many cases, we are interested in knowing whether an event

(sending or receiving a message) at one process occurred before, after or concurrently

with another event at another process. The execution of a system can be described in

terms of events and their ordering despite the lack of accurate clocks.

For example, consider the following set of exchanges between a group of email

users, X, Y, Z and A, on a mailing list:

1. User X sends a message with the subject Meeting.

2. Users Y and Z reply by sending a message with the subject Re: Meeting.

In real time, X’s message is sent first, and Y reads it and replies; Z then reads both X’s

message and Y’s reply and sends another reply, which references both X’s and Y’s

messages. But due to the independent delays in message delivery, the messages may be

delivered as shown in Figure 2.13, and some users may view these two messages in the

wrong order. For example, user A might see:

Inbox:

Item From Subject

23

24

25

ZXY

Re: Meeting

Meeting

Re: Meeting

Figure 2.13 Real-time ordering of events

send

receive

send

receive

m1 m2

2

1

3

X 4

Y Z

Physical

time

A

m3

receive receive

send

receive receive receive

t1 t2 t3

receive

receive

m2

m1SECTION 2.4 FUNDAMENTAL MODELS 83

If the clocks on X’s, Y’s and Z’s computers could be synchronized, then each message

could carry the time on the local computer’s clock when it was sent. For example,

messages m1, m2 and m3 would carry times t1, t2 and t3 where t1<t2<t3. The messages

received will be displayed to users according to their time ordering. If the clocks are

roughly synchronized, then these timestamps will often be in the correct order.

Since clocks cannot be synchronized perfectly across a distributed system,

Lamport [1978] proposed a model of logical time that can be used to provide an ordering

among the events at processes running in different computers in a distributed system.

Logical time allows the order in which the messages are presented to be inferred without

recourse to clocks. It is presented in detail in Chapter 14, but we suggest here how some

aspects of logical ordering can be applied to our email ordering problem.

Logically, we know that a message is received after it was sent. Therefore we can

state a logical ordering for pairs of events shown in Figure 2.13, for example,

considering only the events concerning X and Y:

X sends m1 before Y receives m1; Y sends m2 before X receives m2.

We also know that replies are sent after receiving messages, so we have the following

logical ordering for Y:

Y receives m1 before sending m2.

Logical time takes this idea further by assigning a number to each event corresponding

to its logical ordering, so that later events have higher numbers than earlier ones. For

example, Figure 2.13 shows the numbers 1 to 4 on the events at X and Y.

2.4.2 Failure model

In a distributed system both processes and communication channels may fail – that is,

they may depart from what is considered to be correct or desirable behaviour. The failure

model defines the ways in which failure may occur in order to provide an understanding

of the effects of failures. Hadzilacos and Toueg [1994] provide a taxonomy that

distinguishes between the failures of processes and communication channels. These are

presented under the headings omission failures, arbitrary failures and timing failures.

The failure model will be used throughout the book. For example:

• In Chapter 4, we present the Java interfaces to datagram and stream

communication, which provide different degrees of reliability.

• Chapter 5 presents the request-reply protocol, which supports RMI. Its failure

characteristics depend on the failure characteristics of both processes and

communication channels. The protocol can be built from either datagram or

stream communication. The choice may be decided according to a consideration

of simplicity of implementation, performance and reliability.

• Chapter 17 presents the two-phase commit protocol for transactions. It is designed

to complete in the face of well-defined failures of processes and communication

channels.

Omission failures • The faults classified as omission failures refer to cases when a

process or communication channel fails to perform actions that it is supposed to do.84 CHAPTER 2 SYSTEM MODELS

Process omission failures: The chief omission failure of a process is to crash. When we

say that a process has crashed we mean that it has halted and will not execute any further

steps of its program ever. The design of services that can survive in the presence of faults

can be simplified if it can be assumed that the services on which they depend crash

cleanly – that is, their processes either function correctly or else stop. Other processes

may be able to detect such a crash by the fact that the process repeatedly fails to respond

to invocation messages. However, this method of crash detection relies on the use of

timeouts – that is, a method in which one process allows a fixed period of time for

something to occur. In an asynchronous system a timeout can indicate only that a

process is not responding – it may have crashed or may be slow, or the messages may

not have arrived.

A process crash is called fail-stop if other processes can detect certainly that the

process has crashed. Fail-stop behaviour can be produced in a synchronous system if the

processes use timeouts to detect when other processes fail to respond and messages are

guaranteed to be delivered. For example, if processes p and q are programmed for q to

reply to a message from p, and if process p has received no reply from process q in a

maximum time measured on p’s local clock, then process p may conclude that process

q has failed. The box opposite illustrates the difficulty of detecting failures in an

asynchronous system or of reaching agreement in the presence of failures.

Communication omission failures: Consider the communication primitives send and

receive. A process p performs a send by inserting the message m in its outgoing message

buffer. The communication channel transports m to q’s incoming message buffer.

Process q performs a receive by taking m from its incoming message buffer and

delivering it (see Figure 2.14). The outgoing and incoming message buffers are typically

provided by the operating system.

The communication channel produces an omission failure if it does not transport

a message from p’s outgoing message buffer to q’s incoming message buffer. This is

known as ‘dropping messages’ and is generally caused by lack of buffer space at the

receiver or at an intervening gateway, or by a network transmission error, detected by a

checksum carried with the message data. Hadzilacos and Toueg [1994] refer to the loss

of messages between the sending process and the outgoing message buffer as sendomission failures, to loss of messages between the incoming message buffer and the

receiving process as receive-omission failures, and to loss of messages in between as

channel omission failures. The omission failures are classified together with arbitrary

failures in Figure 2.15.

Failures can be categorized according to their severity. All of the failures we have

described so far are benign failures. Most failures in distributed systems are benign.

Benign failures include failures of omission as well as timing failures and performance

failures.

Arbitrary failures • The term arbitrary or Byzantine failure is used to describe the worst

possible failure semantics, in which any type of error may occur. For example, a process

may set wrong values in its data items, or it may return a wrong value in response to an

invocation.

An arbitrary failure of a process is one in which it arbitrarily omits intended

processing steps or takes unintended processing steps. Arbitrary failures in processesSECTION 2.4 FUNDAMENTAL MODELS 85

cannot be detected by seeing whether the process responds to invocations, because it

might arbitrarily omit to reply.

Communication channels can suffer from arbitrary failures; for example, message

contents may be corrupted, nonexistent messages may be delivered or real messages

may be delivered more than once. Arbitrary failures of communication channels are rare

Failure detection • In the case of the Pepperland divisions encamped at the tops of

hills (see page 81), suppose that the Blue Meanies are after all sufficient in strength

to attack and defeat either division while encamped – that is, that either can fail.

Suppose further that, while undefeated, the divisions regularly send messengers to

report their status. In an asynchronous system, neither division can distinguish

whether the other has been defeated or the time it is taking for the messengers to cross

the intervening valley is just very long. In a synchronous Pepperland, a division can

tell for sure if the other has been defeated by the absence of a regular messenger.

However, the other division may have been defeated just after it sent the latest

messenger.

Impossibility of reaching timely agreement in the presence of communication

failures • We have been assuming that the Pepperland messengers always manage

to cross the valley eventually; but now suppose that the Blue Meanies can capture any

messenger and prevent them from arriving. (We shall assume it is impossible for the

Blue Meanies to brainwash the messengers to give the wrong message – the Meanies

are not aware of their treacherous Byzantine precursors.) Can the Apple and Orange

divisions send messages so that they both consistently decide to charge at the

Meanies or both decide to surrender? Unfortunately, as the Pepperland theoretician

Ringo the Great proved, in these circumstances the divisions cannot guarantee to

decide consistently what to do. To see this, assume to the contrary that the divisions

run a Pepperland protocol that achieves agreement. Each proposes ‘Charge!’ or

‘Surrender!’, and the protocol results in them both agreeing on one or the other course

of action. Now consider the last message sent in any run of the protocol. The

messenger that carries it could be captured by the Blue Meanies, so the end result

must be the same whether the message arrives or not. We can dispense with it. Now

we can apply the same argument to the final message that remains. But this argument

applies again to that message and will continue to apply, so we shall end up with no

messages sent at all! This shows that no protocol that guarantees agreement between

the Pepperland divisions can exist if messengers can be captured.

Figure 2.14 Processes and channels

process p process q

Communication channel

send

Outgoing message buffer Incoming message buffer

m receive86 CHAPTER 2 SYSTEM MODELS

because the communication software is able to recognize them and reject the faulty

messages. For example, checksums are used to detect corrupted messages, and message

sequence numbers can be used to detect nonexistent and duplicated messages.

Timing failures • Timing failures are applicable in synchronous distributed systems

where time limits are set on process execution time, message delivery time and clock

drift rate. Timing failures are listed in Figure 2.16. Any one of these failures may result

in responses being unavailable to clients within a specified time interval.

In an asynchronous distributed system, an overloaded server may respond too

slowly, but we cannot say that it has a timing failure since no guarantee has been offered.

Real-time operating systems are designed with a view to providing timing

guarantees, but they are more complex to design and may require redundant hardware.

Most general-purpose operating systems such as UNIX do not have to meet real-time

constraints.

Timing is particularly relevant to multimedia computers with audio and video

channels. Video information can require a very large amount of data to be transferred.

Delivering such information without timing failures can make very special demands on

both the operating system and the communication system.

Masking failures • Each component in a distributed system is generally constructed

from a collection of other components. It is possible to construct reliable services from

components that exhibit failures. For example, multiple servers that hold replicas of data

can continue to provide a service when one of them crashes. A knowledge of the failure

characteristics of a component can enable a new service to be designed to mask the

Figure 2.15 Omission and arbitrary failures

Class of failure Affects Description

Fail-stop Process Process halts and remains halted. Other processes may

detect this state.

Crash Process Process halts and remains halted. Other processes may

not be able to detect this state.

Omission Channel A message inserted in an outgoing message buffer

never arrives at the other end’s incoming message

buffer.

Send-omission Process A process completes a send operation but the message

is not put in its outgoing message buffer.

Receiveomission

Process A message is put in a process’s incoming message

buffer, but that process does not receive it.

Arbitrary

(Byzantine)

Process

or

channel

Process/channel exhibits arbitrary behaviour: it may

send/transmit arbitrary messages at arbitrary times or

commit omissions; a process may stop or take an

incorrect step.SECTION 2.4 FUNDAMENTAL MODELS 87

failure of the components on which it depends. A service masks a failure either by hiding

it altogether or by converting it into a more acceptable type of failure. For an example

of the latter, checksums are used to mask corrupted messages, effectively converting an

arbitrary failure into an omission failure. We shall see in Chapters 3 and 4 that omission

failures can be hidden by using a protocol that retransmits messages that do not arrive at

their destination. Chapter 18 presents masking by means of replication. Even process

crashes may be masked, by replacing the process and restoring its memory from

information stored on disk by its predecessor.

Reliability of one-to-one communication • Although a basic communication channel

can exhibit the omission failures described above, it is possible to use it to build a

communication service that masks some of those failures.

The term reliable communication is defined in terms of validity and integrity as

follows:

Validity: Any message in the outgoing message buffer is eventually delivered to the

incoming message buffer.

Integrity: The message received is identical to one sent, and no messages are

delivered twice.

The threats to integrity come from two independent sources:

• Any protocol that retransmits messages but does not reject a message that arrives

twice. Protocols can attach sequence numbers to messages so as to detect those

that are delivered twice.

• Malicious users that may inject spurious messages, replay old messages or tamper

with messages. Security measures can be taken to maintain the integrity property

in the face of such attacks.

2.4.3 Security model

In Chapter 1 we identified the sharing of resources as a motivating factor for distributed

systems, and in Section 2.3 we described their architecture in terms of processes,

potentially encapsulating higher-level abstractions such as objects, components or

Figure 2.16 Timing failures

Class of failure Affects Description

Clock Process Process’s local clock exceeds the bounds on

its rate of drift from real time.

Performance Process Process exceeds the bounds on the interval

between two steps.

Performance Channel A message’s transmission takes longer than

the stated bound.88 CHAPTER 2 SYSTEM MODELS

services, and providing access to them through interactions with other processes. That

architectural model provides the basis for our security model:

the security of a distributed system can be achieved by securing the processes and the

channels used for their interactions and by protecting the objects that they

encapsulate against unauthorized access.

Protection is described in terms of objects, although the concepts apply equally well to

resources of all types.

Protecting objects • Figure 2.17 shows a server that manages a collection of objects on

behalf of some users. The users can run client programs that send invocations to the

server to perform operations on the objects. The server carries out the operation

specified in each invocation and sends the result to the client.

Objects are intended to be used in different ways by different users. For example,

some objects may hold a user’s private data, such as their mailbox, and other objects

may hold shared data such as web pages. To support this, access rights specify who is

allowed to perform the operations of an object – for example, who is allowed to read or

to write its state.

Thus we must include users in our model as the beneficiaries of access rights. We

do so by associating with each invocation and each result the authority on which it is

issued. Such an authority is called a principal. A principal may be a user or a process.

In our illustration, the invocation comes from a user and the result from a server.

The server is responsible for verifying the identity of the principal behind each

invocation and checking that they have sufficient access rights to perform the requested

operation on the particular object invoked, rejecting those that do not. The client may

check the identity of the principal behind the server to ensure that the result comes from

the required server.

Securing processes and their interactions • Processes interact by sending messages.

The messages are exposed to attack because the network and the communication service

that they use are open, to enable any pair of processes to interact. Servers and peer

processes expose their interfaces, enabling invocations to be sent to them by any other

process.

Distributed systems are often deployed and used in tasks that are likely to be

subject to external attacks by hostile users. This is especially true for applications that

Figure 2.17 Objects and principals

Network

invocation

result

Client

Server

Principal (user) Principal (server)

Access rights ObjectSECTION 2.4 FUNDAMENTAL MODELS 89

handle financial transactions, confidential or classified information or any other

information whose secrecy or integrity is crucial. Integrity is threatened by security

violations as well as communication failures. So we know that there are likely to be

threats to the processes of which such applications are composed and to the messages

travelling between the processes. But how can we analyze these threats in order to

identify and defeat them? The following discussion introduces a model for the analysis

of security threats.

The enemy • To model security threats, we postulate an enemy (sometimes also known

as the adversary) that is capable of sending any message to any process and reading or

copying any message sent between a pair of processes, as shown in Figure 2.18. Such

attacks can be made simply by using a computer connected to a network to run a

program that reads network messages addressed to other computers on the network, or

a program that generates messages that make false requests to services, purporting to

come from authorized users. The attack may come from a computer that is legitimately

connected to the network or from one that is connected in an unauthorized manner.

The threats from a potential enemy include threats to processes and threats to

communication channels.

Threats to processes: A process that is designed to handle incoming requests may

receive a message from any other process in the distributed system, and it cannot

necessarily determine the identity of the sender. Communication protocols such as IP do

include the address of the source computer in each message, but it is not difficult for an

enemy to generate a message with a forged source address. This lack of reliable

knowledge of the source of a message is a threat to the correct functioning of both

servers and clients, as explained below:

Servers: Since a server can receive invocations from many different clients, it cannot

necessarily determine the identity of the principal behind any particular invocation.

Even if a server requires the inclusion of the principal’s identity in each invocation,

an enemy might generate an invocation with a false identity. Without reliable

knowledge of the sender’s identity, a server cannot tell whether to perform the

operation or to reject it. For example, a mail server would not know whether the user

behind an invocation that requests a mail item from a particular mailbox is allowed

to do so or whether it was a request from an enemy.

Clients: When a client receives the result of an invocation from a server, it cannot

necessarily tell whether the source of the result message is from the intended server

Figure 2.18 The enemy

Communication channel

Copy of m

Process p m Process q

m'

The enemy90 CHAPTER 2 SYSTEM MODELS

or from an enemy, perhaps ‘spoofing’ the mail server. Thus the client could receive

a result that was unrelated to the original invocation, such as a false mail item (one

that is not in the user’s mailbox).

Threats to communication channels: An enemy can copy, alter or inject messages as they

travel across the network and its intervening gateways. Such attacks present a threat to

the privacy and integrity of information as it travels over the network and to the integrity

of the system. For example, a result message containing a user’s mail item might be

revealed to another user or it might be altered to say something quite different.

Another form of attack is the attempt to save copies of messages and to replay

them at a later time, making it possible to reuse the same message over and over again.

For example, someone could benefit by resending an invocation message requesting a

transfer of a sum of money from one bank account to another.

All these threats can be defeated by the use of secure channels, which are

described below and are based on cryptography and authentication.

Defeating security threats • Here we introduce the main techniques on which secure

systems are based. Chapter 11 discusses the design and implementation of secure

distributed systems in much more detail.

Cryptography and shared secrets: Suppose that a pair of processes (for example, a

particular client and a particular server) share a secret; that is, they both know the secret

but no other process in the distributed system knows it. Then if a message exchanged by

that pair of processes includes information that proves the sender’s knowledge of the

shared secret, the recipient knows for sure that the sender was the other process in the

pair. Of course, care must be taken to ensure that the shared secret is not revealed to an

enemy.

Cryptography is the science of keeping messages secure, and encryption is the

process of scrambling a message in such a way as to hide its contents. Modern

cryptography is based on encryption algorithms that use secret keys – large numbers that

are difficult to guess – to transform data in a manner that can only be reversed with

knowledge of the corresponding decryption key.

Authentication: The use of shared secrets and encryption provides the basis for the

authentication of messages – proving the identities supplied by their senders. The basic

authentication technique is to include in a message an encrypted portion that contains

enough of the contents of the message to guarantee its authenticity. The authentication

portion of a request to a file server to read part of a file, for example, might include a

representation of the requesting principal’s identity, the identity of the file and the date

and time of the request, all encrypted with a secret key shared between the file server

and the requesting process. The server would decrypt this and check that it corresponds

to the unencrypted details specified in the request.

Secure channels: Encryption and authentication are used to build secure channels as a

service layer on top of existing communication services. A secure channel is a

communication channel connecting a pair of processes, each of which acts on behalf of

a principal, as shown in Figure 2.19. A secure channel has the following properties:

• Each of the processes knows reliably the identity of the principal on whose behalf

the other process is executing. Therefore if a client and server communicate via a

secure channel, the server knows the identity of the principal behind theSECTION 2.4 FUNDAMENTAL MODELS 91

invocations and can check their access rights before performing an operation. This

enables the server to protect its objects correctly and allows the client to be sure

that it is receiving results from a bona fide server.

• A secure channel ensures the privacy and integrity (protection against tampering)

of the data transmitted across it.

• Each message includes a physical or logical timestamp to prevent messages from

being replayed or reordered.

The construction of secure channels is discussed in detail in Chapter 11. Secure channels

have become an important practical tool for securing electronic commerce and the

protection of communication. Virtual private networks (VPNs, discussed in Chapter 3)

and the Secure Sockets Layer (SSL) protocol (discussed in Chapter 11) are instances.

Other possible threats from an enemy • Section 1.5.3 introduced very briefly two

further security threats – denial of service attacks and the deployment of mobile code.

We reiterate these as possible opportunities for the enemy to disrupt the activities of

processes:

Denial of service: This is a form of attack in which the enemy interferes with the

activities of authorized users by making excessive and pointless invocations on

services or message transmissions in a network, resulting in overloading of physical

resources (network bandwidth, server processing capacity). Such attacks are usually

made with the intention of delaying or preventing actions by other users. For

example, the operation of electronic door locks in a building might be disabled by an

attack that saturates the computer controlling the electronic locks with invalid

requests.

Mobile code: Mobile code raises new and interesting security problems for any

process that receives and executes program code from elsewhere, such as the email

attachment mentioned in Section 1.5.3. Such code may easily play a Trojan horse

role, purporting to fulfil an innocent purpose but in fact including code that accesses

or modifies resources that are legitimately available to the host process but not to the

originator of the code. The methods by which such attacks might be carried out are

many and varied, and the host environment must be very carefully constructed in

order to avoid them. Many of these issues have been addressed in Java and other

mobile code systems, but the recent history of this topic has included the exposure of

Figure 2.19 Secure channels

Principal A

Process p Secure channel Process q

Principal B92 CHAPTER 2 SYSTEM MODELS

some embarrassing weaknesses. This illustrates well the need for rigorous analysis in

the design of all secure systems.

The uses of security models • It might be thought that the achievement of security in

distributed systems would be a straightforward matter involving the control of access to

objects according to predefined access rights and the use of secure channels for

communication. Unfortunately, this is not generally the case. The use of security

techniques such as encryption and access control incurs substantial processing and

management costs. The security model outlined above provides the basis for the analysis

and design of secure systems in which these costs are kept to a minimum, but threats to

a distributed system arise at many points, and a careful analysis of the threats that might

arise from all possible sources in the system’s network environment, physical

environment and human environment is needed. This analysis involves the construction

of a threat model listing all the forms of attack to which the system is exposed and an

evaluation of the risks and consequences of each. The effectiveness and the cost of the

security techniques needed can then be balanced against the threats.

2.5 Summary

As illustrated in Section 2.2, distributed systems are increasingly complex in terms of

their underlying physical characteristics; for example, in terms of the scale of systems,

the level of heterogeneity inherent in such systems and the real demands to provide endto-end solutions in terms of properties such as security. This places increasing

importance on being able to understand and reason about distributed systems in terms of

models. This chapter followed up consideration of the underlying physical models with

an in-depth examination of the architectural and fundamental models that underpin

distributed systems.

This chapter has presented an approach to describing distributed systems in terms

of an encompassing architectural model that makes sense of this design space examining

the core issues of what is communicating and how these entities communicate,

supplemented by consideration of the roles each element may play together with the

appropriate placement strategies given the physical distributed infrastructure. The

chapter also introduced the key role of architectural patterns in enabling more complex

designs to be constructed from the underlying core elements, such as the client-server

model highlighted above, and highlighted major styles of supportive middleware

solutions, including solutions based on distributed objects, components, web services

and distributed events.

In terms of architectural models, the client-server approach is prevalent – the Web

and other Internet services such as FTP, news and mail as well as web services and the

DNS are based on this model, as are filing and other local services. Services such as the

DNS that have large numbers of users and manage a great deal of information are based

on multiple servers and use data partition and replication to enhance availability and

fault tolerance. Caching by clients and proxy servers is widely used to enhance the

performance of a service. However, there is now a wide variety of approaches to

modelling distributed systems including alternative philosophies such as peer-to-peerEXERCISES 93

computing and support for more problem-oriented abstractions such as objects,

components or services.

The architectural model is complemented by fundamental models, which aid in

reasoning about properties of the distributed system in terms of, for example,

performance, reliability and security. In particular, we presented models of interaction,

failure and security. They identify the common characteristics of the basic components

from which distributed systems are constructed. The interaction model is concerned

with the performance of processes and communication channels and the absence of a

global clock. It identifies a synchronous system as one in which known bounds may be

placed on process execution time, message delivery time and clock drift. It identifies an

asynchronous system as one in which no bounds may be placed on process execution

time, message delivery time and clock drift – which is a description of the behaviour of

the Internet.

The failure model classifies the failures of processes and basic communication

channels in a distributed system. Masking is a technique by which a more reliable

service is built from a less reliable one by masking some of the failures it exhibits. In

particular, a reliable communication service can be built from a basic communication

channel by masking its failures. For example, its omission failures may be masked by

retransmitting lost messages. Integrity is a property of reliable communication – it

requires that a message received be identical to one that was sent and that no message

be sent twice. Validity is another property – it requires that any message put in the

outgoing buffer be delivered eventually to the incoming message buffer.

The security model identifies the possible threats to processes and communication

channels in an open distributed system. Some of those threats relate to integrity:

malicious users may tamper with messages or replay them. Others threaten their privacy.

Another security issue is the authentication of the principal (user or server) on whose

behalf a message was sent. Secure channels use cryptographic techniques to ensure the

integrity and privacy of messages and to authenticate pairs of communicating principals.

EXERCISES

2.1 What is the main disadvantage of distributed systems which exploit the infrastructure

offered by the Internet? How can this be overcome? page 55

2.2 What problems do you foresee in the direct coupling between communicating entities

that is implicit in remote invocation approaches? Consequently, what advantages do you

anticipate from a level of decoupling as offered by space and time uncoupling? Note:

you might want to revisit this answer after reading Chapters 5 and 6. page 59

2.3 What is the range of techniques covered by remote invocation? Briefly explain each

technique. page 60

2.4 How are entities, such as objects or services, mapped on to the underlying physical

distributed infrastructure? page 6494 CHAPTER 2 SYSTEM MODELS

2.5 A search engine is a web server that responds to client requests to search in its stored

indexes and (concurrently) runs several web crawler tasks to build and update the

indexes. What are the requirements for synchronization between these concurrent

activities? page 62

2.6 The host computers used in peer-to-peer systems are often simply desktop computers in

users’ offices or homes. What are the implications of this for the availability and security

of any shared data objects that they hold and to what extent can any weaknesses be

overcome through the use of replication? pages 63, 64

2.7 How is caching useful in placement strategies? What are its disadvantages? page 65

2.8 What is a mobile agent? How can it be a potential security threat? page 67

2.9 Consider a hypothetical car hire company and sketch out a three-tier solution to the

provision of their underlying distributed car hire service. Use this to illustrate the

benefits and drawbacks of a three-tier solution considering issues such as performance,

scalability, dealing with failure and also maintaining the software over time. page 68

2.10 Provide a concrete example of the dilemma offered by Saltzer’s end-to-end argument in

the context of the provision of middleware support for distributed applications (you may

want to focus on one aspect of providing dependable distributed systems, for example

related to fault tolerance or security). page 76

2.11 Consider a simple server that carries out client requests without accessing other servers.

Explain why it is generally not possible to set a limit on the time taken by such a server

to respond to a client request. What would need to be done to make the server able to

execute requests within a bounded time? Is this a practical option? page 78

2.12 For each of the factors that contribute to the time taken to transmit a message between

two processes over a communication channel, state what measures would be needed to

set a bound on its contribution to the total time. Why are these measures not provided in

current general-purpose distributed systems? page 79

2.13 What are the two variants of the interaction model in distributed systems? On what

points do they differ? page 80

2.14 Consider two communication services for use in asynchronous distributed systems. In

service A, messages may be lost, duplicated or delayed and checksums apply only to

headers. In service B, messages may be lost, delayed or delivered too fast for the

recipient to handle them, but those that are delivered arrive with the correct contents.

Describe the classes of failure exhibited by each service. Classify their failures

according to their effects on the properties of validity and integrity. Can service B be

described as a reliable communication service? page 83, page 87

2.15 Consider a pair of processes X and Y that use the communication service B from

Exercise 2.14 to communicate with one another. Suppose that X is a client and Y a

server and that an invocation consists of a request message from X to Y, followed by Y

carrying out the request, followed by a reply message from Y to X. Describe the classes

of failure that may be exhibited by an invocation. page 83EXERCISES 95

2.16 Suppose that a basic disk read can sometimes read values that are different from those

written. State the type of failure exhibited by a basic disk read. Suggest how this failure

may be masked in order to produce a different benign form of failure. Now suggest how

to mask the benign failure. page 86

2.17 How can the security of a distributed system be achieved? How can processes and their

interactions be secured? pages 88, 89

2.18 Cryptography is the science of keeping messages secure. Explain, with an example,

how it can be used in authentication to maintain confidentiality. pages 90, 91This page intentionally left blank97

3

NETWORKING AND

INTERNETWORKING

3.1 Introduction

3.2 Types of network

3.3 Network principles

3.4 Internet protocols

3.5 Case studies: Ethernet, WiFi and Bluetooth

3.6 Summary

Distributed systems use local area networks, wide area networks and internetworks for

communication. The performance, reliability, scalability, mobility and quality of service

characteristics of the underlying networks impact the behaviour of distributed systems

and hence affect their design. Changes in user requirements have resulted in the

emergence of wireless networks and of high-performance networks with quality of service

guarantees.

The principles on which computer networks are based include protocol layering,

packet switching, routing and data streaming. Internetworking techniques enable

heterogeneous networks to be integrated. The Internet is the major example; its protocols

are almost universally used in distributed systems. The addressing and routing schemes

used in the Internet have withstood the impact of its enormous growth. They are now

undergoing revision to accommodate future growth and to meet new application

requirements for mobility, security and quality of service.

The design of specific network technologies is illustrated in three case studies:

Ethernet, IEEE 802.11 (WiFi) and Bluetooth wireless networking.98 CHAPTER 3 NETWORKING AND INTERNETWORKING

3.1 Introduction

The networks used in distributed systems are built from a variety of transmission media,

including wire, cable, fibre and wireless channels; hardware devices, including routers,

switches, bridges, hubs, repeaters and network interfaces; and software components,

including protocol stacks, communication handlers and drivers. The resulting

functionality and performance available to distributed system and application programs

is affected by all of these. We shall refer to the collection of hardware and software

components that provide the communication facilities for a distributed system as a

communication subsystem. The computers and other devices that use the network for

communication purposes are referred to as hosts. The term node is used to refer to any

computer or switching device attached to a network.

The Internet is a single communication subsystem providing communication

between all of the hosts that are connected to it. The Internet is constructed from many

subnets. A subnet is a unit of routing (delivering data from one part of the Internet to

another); it is a collection of nodes that can all be reached on the same physical network.

The Internet’s infrastructure includes an architecture and hardware and software

components that effectively integrate diverse subnets into a single data communication

service.

The design of a communication subsystem is strongly influenced by the

characteristics of the operating systems used in the computers of which the distributed

system is composed as well as the networks that interconnect them. In this chapter, we

consider the impact of network technologies on the communication subsystem;

operating system issues are discussed in Chapter 7.

This chapter is intended to provide an introductory overview of computer

networking with reference to the communication requirements of distributed systems.

Readers who are not familiar with computer networking should regard it as an

underpinning for the remainder of the book, while those who are will find that this

chapter offers an extended summary of those aspects of computer networking that are

particularly relevant for distributed systems.

Computer networking was conceived soon after the invention of computers. The

theoretical basis for packet switching was introduced in a paper by Leonard Kleinrock

[1961]. In 1962, J.C.R. Licklider and W. Clark, who participated in the development of

the first timesharing system at MIT in the early 1960s, published a paper discussing the

potential for interactive computing and wide area networking that presaged the Internet

in several respects [DEC 1990]. In 1964, Paul Baran produced an outline of a practical

design for reliable and effective wide area networks [Baran 1964]. Further material and

links on the history of computer networking and the Internet can be found in the

following sources: [www.isoc.org, Comer 2007, Kurose and Ross 2007].

In the remainder of this section we discuss the communication requirements of

distributed systems. We give an overview of network types in Section 3.2 and an

introduction to networking principles in Section 3.3. Section 3.4 deals specifically with

the Internet. The chapter concludes with detailed case studies on the Ethernet, IEEE

802.11 (WiFi) and Bluetooth networking technologies in Section 3.5.SECTION 3.1 INTRODUCTION 99

3.1.1 Networking issues for distributed systems

Early computer networks were designed to meet a few, relatively simple application

requirements. Network applications such as file transfer, remote login, electronic mail and

newsgroups were supported. The subsequent development of distributed systems with

support for distributed application programs accessing shared files and other resources set

a higher standard of performance to meet the needs of interactive applications.

More recently, following the growth and commercialization of the Internet and the

emergence of many new modes of use, more stringent requirements for reliability,

scalability, mobility, security and quality of service have emerged. In this section, we

define and describe the nature of each of these requirements.

Performance • The network performance parameters that are of primary interest for our

purposes are those affecting the speed with which individual messages can be

transferred between two interconnected computers. These are the latency and the pointto-point data transfer rate:

Latency is the delay that occurs after a send operation is executed and before data

starts to arrive at the destination computer. It can be measured as the time required to

transfer an empty message. Here we are considering only network latency, which

forms a part of the process-to-process latency defined in Section 2.4.1.

Data transfer rate is the speed at which data can be transferred between two

computers in the network once transmission has begun, usually quoted in bits per

second.

Following from these definitions, the time required for a network to transfer a message

containing length bits between two computers is:

Message transmission time = latency + length ⁄ data transfer rate

The above equation is valid for messages whose length does not exceed a maximum that

is determined by the underlying network technology. Longer messages have to be

segmented and the transmission time is the sum of the times for the segments.

The transfer rate of a network is determined primarily by its physical

characteristics, whereas the latency is determined primarily by software overheads,

routing delays and a load-dependent statistical element arising from conflicting

demands for access to transmission channels. Many of the messages transferred between

processes in distributed systems are small in size; latency is therefore often of equal or

greater significance than transfer rate in determining performance.

The total system bandwidth of a network is a measure of throughput – the total

volume of traffic that can be transferred across the network in a given time. In many

local area network technologies, such as Ethernet, the full transmission capacity of the

network is used for every transmission and the system bandwidth is the same as the data

transfer rate. But in most wide area networks messages can be transferred on several

different channels simultaneously, and the total system bandwidth bears no direct

relationship to the transfer rate. The performance of networks deteriorates in conditions

of overload – when there are too many messages in the network at the same time. The

precise effect of overload on the latency, data transfer rate and total system bandwidth

of a network depends strongly on the network technology.100 CHAPTER 3 NETWORKING AND INTERNETWORKING

Now consider the performance of client-server communication. The time required

to transmit a short request message and receive a short reply between nodes on a lightly

loaded local network (including system overheads) is about half a millisecond. This

should be compared with the sub-microsecond time required to invoke an operation on

an application-level object in the local memory. Thus, despite advances in network

performance, the time required to access shared resources on a local network remains

about a thousand times greater than that required to access resources that are resident in

local memory. But networks often outperform hard disks; networked access to a local

web server or file server with a large in-memory cache of frequently used files can match

or outstrip access to files stored on a local hard disk.

On the Internet, round-trip latencies are in the 5–500 ms range, with means of 20–

200 ms depending on distance [www.globalcrossing.net], so requests transmitted

across the Internet are 10–100 times slower than those sent on fast local networks. The

bulk of this time difference derives from switching delays at routers and contention for

network circuits.

Section 7.5.1 discusses and compares the performance of local and remote

operations in greater detail.

Scalability • Computer networks are an indispensable part of the infrastructure of

modern societies. In Figure 1.6 we showed the growth in the number of host computers

and web servers connected to the Internet over a 12-year period ending in 2005. The

growth since then has been so rapid and diverse that it is difficult to find recent reliable

statistics. The potential future size of the Internet is commensurate with the population

of the planet. It is realistic to expect it to include several billion nodes and hundreds of

millions of active hosts.

These numbers indicate the future changes in size and load that the Internet must

handle. The network technologies on which it is based were not designed to cope with

even the Internet’s current scale, but they have performed remarkably well. Some

substantial changes to the addressing and routing mechanisms are in progress in order

to handle the next phase of the Internet’s growth; these will be described in Section 3.4.

For simple client-server applications such as the Web, we would expect future traffic to

grow at least in proportion to the number of active users. The ability of the Internet’s

infrastructure to cope with this growth will depend upon the economics of use, in

particular charges to users and the patterns of communication that actually occur – for

example, their degree of locality.

Reliability • Our discussion of failure models in Section 2.4.2 describes the impact of

communication errors. Many applications are able to recover from communication

failures and hence do not require guaranteed error-free communication. The end-to-end

argument (Section 2.3.3) further supports the view that the communication subsystem

need not provide totally error-free communication; the detection of communication

errors and their correction is often best performed by application-level software. The

reliability of most physical transmission media is very high. When errors occur they

are usually due to failures in the software at the sender or receiver (for example,

failure by the receiving computer to accept a packet) or buffer overflow rather than

errors in the network.SECTION 3.1 INTRODUCTION 101

Security • Chapter 11 sets out the requirements and techniques for achieving security

in distributed systems. The first level of defence adopted by most organizations is to

protect its networks and the computers attached to them with a firewall. A firewall

creates a protection boundary between the organization’s intranet and the rest of the

Internet. The purpose of the firewall is to protect the resources in all of the computers

inside the organization from access by external users or processes and to control the use

of resources outside the firewall by users inside the organization.

A firewall runs on a gateway – a computer that stands at the network entry point

to an organization’s intranet. The firewall receives and filters all of the messages

travelling into and out of an organization. It is configured according to the

organization’s security policy to allow certain incoming and outgoing messages to pass

through it and to reject all others. We shall return to this topic in Section 3.4.8.

To enable distributed applications to move beyond the restrictions imposed by

firewalls there is a need to produce a secure network environment in which a wide range

of distributed applications can be deployed, with end-to-end authentication, privacy and

security. This finer-grained and more flexible form of security can be achieved through

the use of cryptographic techniques. It is usually applied at a level above the

communication subsystem and hence is not dealt with here but in Chapter 11.

Exceptions include the need to protect network components such as routers against

unauthorized interference with their operation and the need for secure links to mobile

devices and other external nodes to enable them to participate in a secure intranet – the

virtual private network (VPN) concept, discussed in Section 3.4.8.

Mobility • Mobile devices such as laptop computers and Internet-capable mobile

phones are moved frequently between locations and reconnected at convenient network

connection points or even used while on the move. Wireless networks provide

connectivity to such devices, but the addressing and routing schemes of the Internet

were developed before the advent of these mobile devices and are not well adapted to

their need for intermittent connection to many different subnets. The Internet’s

mechanisms have been adapted and extended to support mobility, but the expected

future growth in the use of mobile devices will demand further development.

Quality of service • In Chapter 1, we defined quality of service as including the ability

to meet deadlines when transmitting and processing streams of real-time multimedia

data. This imposes major new requirements on computer networks. Applications that

transmit multimedia data require guaranteed bandwidth and bounded latencies for the

communication channels that they use. Some applications vary their demands

dynamically and specify both a minimum acceptable quality of service and a desired

optimum. The provision of such guarantees and their maintenance is the subject of

Chapter 20.

Multicasting • Most communication in distributed systems is between pairs of

processes, but there often is also a need for one-to-many communication. While this can

be simulated by sends to several destinations, that is more costly than necessary and may

not exhibit the fault-tolerance characteristics required by applications. For these reasons,

many network technologies support the simultaneous transmission of messages to

several recipients.102 CHAPTER 3 NETWORKING AND INTERNETWORKING

3.2 Types of network

Here we introduce the main types of network that are used to support distributed

systems: personal area networks, local area networks, wide area networks,

metropolitan area networks and the wireless variants of them. Internetworks such as the

Internet are constructed from networks of all these types. Figure 3.1 shows the

performance characteristics of the various types of network discussed below.

Some of the names used to refer to types of networks are confusing because they

seem to refer to the physical extent (local area, wide area), but they also identify physical

transmission technologies and low-level protocols. These are different for local and

wide area networks, although some network technologies, such as ATM (Asynchronous

Transfer Mode), are suitable for both local and wide area applications and some wireless

networks also support local and metropolitan area transmission.

We refer to networks that are composed of many interconnected networks,

integrated to provide a single data communication medium, as internetworks. The

Internet is the prototypical internetwork; it is composed of millions of local,

metropolitan and wide area networks. We describe its implementation in some detail in

Section 3.4.

Personal area networks (PANs) • PANs are a subcategory of local networks in which

the various digital devices carried by a user are connected by a low-cost, low-energy

network. Wired PANs are not of much significance because few users wish to be

encumbered by a network of wires on their person, but wireless personal area networks

(WPANs) are of increasing importance due to the number of personal devices such as

mobile phones, tablets, digital cameras, music players and so on that are now carried by

many people. We describe the Bluetooth WPAN in Section 3.5.3.

Local area networks (LANs) • LANs carry messages at relatively high speeds between

computers connected by a single communication medium, such as twisted copper wire,

Figure 3.1 Network performance

Example Range Bandwidth

(Mbps)

Latency

(ms)

Wired:

LAN Ethernet 1–2 kms 10–10,000 1–10

WAN IP routing worldwide 0.010–600 100–500

MAN ATM 2–50 kms 1–600 10

Internetwork Internet worldwide 0.5–600 100–500

Wireless:

WPAN Bluetooth (IEEE 802.15.1) 10–30m 0.5–2 5–20

WLAN WiFi (IEEE 802.11) 0.15–1.5 km 11–108 5–20

WMAN WiMAX (IEEE 802.16) 5–50 km 1.5–20 5–20

WWAN 3G phone cell: 1–-5

km

348–14.4 100–500SECTION 3.2 TYPES OF NETWORK 103

coaxial cable or optical fibre. A segment is a section of cable that serves a department or

a floor of a building and may have many computers attached. No routing of messages is

required within a segment, since the medium provides direct connections between all of

the computers connected to it. The total system bandwidth is shared between the

computers connected to a segment. Larger local networks, such as those that serve a

campus or an office building, are composed of many segments interconnected by

switches or hubs (see Section 3.3.7). In local area networks, the total system bandwidth

is high and latency is low, except when message traffic is very high.

Several local area technologies were developed in the 1970s including Ethernet,

token rings and slotted rings. Each provides an effective and high-performance solution,

but Ethernet emerged as the dominant technology for wired local area networks. It was

originally produced in the early 1970s with a bandwidth of 10 Mbps (million bits per

second) and extended to 100 Mbps, 1000 Mbps (1 gigabit per second) and 10 Gbps

versions more recently. We describe the principles of operation of Ethernet networks in

Section 3.5.1.

There is a very large installed base of local area networks, serving virtually all

working environments that contain more than one or two personal computers or

workstations. Their performance is generally adequate for the implementation of

distributed systems and applications. Ethernet technology lacks the latency and

bandwidth guarantees needed by many multimedia applications. ATM networks were

developed to fill this gap, but their cost has inhibited their adoption in local area

applications. Instead, high-speed Ethernets have been deployed in a switched mode that

overcomes these drawbacks to a significant degree, though not as effectively as ATM.

Wide area networks (WANs) • WANs carry messages at lower speeds between nodes

that are often in different organizations and may be separated by large distances. They

may be located in different cities, countries or continents. The communication medium

is a set of communication circuits linking a set of dedicated computers called routers.

They manage the communication network and route messages or packets to their

destinations. In most networks, the routing operations introduce a delay at each point in

the route, so the total latency for the transmission of a message depends on the route that

it follows and the traffic loads in the various network segments that it traverses. In

current networks these latencies can be as high as 0.1 to 0.5 seconds. The speed of

electronic signals in most media is close to the speed of light, and this sets a lower bound

on the transmission latency for long-distance networks. For example, the propagation

delay for a signal to travel from Europe to Australia via a terrestrial link is approximately

0.13 seconds and signals via a geostationary satellite between any two points on the

Earth’s surface are subject to a delay of approximately 0.20 seconds.

Bandwidths available across the Internet also vary widely. Speeds of up to 600

Mbps are commonly available, but speeds of 1–10 Mbps are more typically experienced

for bulk transfers of data.

Metropolitan area networks (MANs) • This type of network is based on the highbandwidth copper and fibre optic cabling recently installed in some towns and cities for

the transmission of video, voice and other data over distances of up to 50 kilometres. A

variety of technologies have been used to implement the routing of data in MANs,

ranging from Ethernet to ATM.104 CHAPTER 3 NETWORKING AND INTERNETWORKING

The DSL (Digital Subscriber Line) and cable modem connections now available

in many countries are an example. DSL typically uses ATM switches located in

telephone exchanges to route digital data onto twisted pairs of copper wire (using highfrequency signalling on the existing wiring used for telephone connections) to the

subscriber’s home or office at speeds in the range 1–10 Mbps. The use of twisted copper

wire for DSL subscriber connections limits the range to about 5.5 km from the switch.

Cable modem connections use analogue signalling on cable television networks to

achieve speeds of up to 15 Mbps over coaxial cable with greater range than DSL.

The term DSL actually represents a family of technologies, sometimes referred to

as xDSL and including for example ADSL (or Asymmetric Digital Subscriber Line).

Latest developments include VDSL and VDSL2 (Very High Bit Rate DSL), which are

capable of speeds of up to 100 Mbps and designed to support a range of multimedia

traffic including High Definition TV (HDTV).

Wireless local area networks (WLANs) • WLANs are designed for use in place of wired

LANs to provide connectivity for mobile devices, or simply to remove the need for a

wired infrastructure to connect computers within homes and office buildings to each

other and the Internet. They are in widespread use in several variants of the IEEE 802.11

standard (WiFi), offering bandwidths of 10–100 Mbps over ranges up to 1.5 kilometres.

Section 3.5.2 gives further information on their method of operation.

Wireless metropolitan area networks (WMANs) • The IEEE 802.16 WiMAX standard is

targeted at this class of network. It aims to provide an alternative to wired connections

to home and office buildings and to supersede 802.11 WiFi networks in some

applications.

Wireless wide area networks (WWANs) • Most mobile phone networks are based on

digital wireless network technologies such as the GSM (Global System for Mobile

communication) standard, which is used in most countries of the world. Mobile phone

networks are designed to operate over wide areas (typically entire countries or

continents) through the use of cellular radio connections; their data transmission

facilities therefore offer wide area mobile connections to the Internet for portable

devices. The cellular networks mentioned above offer relatively low data rates – 9.6 to

33 kbps – but the ‘third generation’ (3G) of mobile phone networks is now available,

with data transmission rates in the range of 2–14.4 Mbps while stationary and 348 kbps

while moving (for example in a car). The underlying technology is referred to as UMTS

(Universal Mobile Telecommunications System). A path has also been defined to evolve

UMTS towards 4G data rates of up to 100 Mbps. Readers interested in digging more

deeply than we are able to here into the rapidly evolving technologies of mobile and

wireless networks of all types are referred to Stojmenovic’s excellent handbook [2002].

Internetworks • An internetwork is a communication subsystem in which several

networks are linked together to provide common data communication facilities that

overlay the technologies and protocols of the individual component networks and the

methods used for their interconnection.

Internetworks are needed for the development of extensible, open distributed

systems. The openness characteristic of distributed systems implies that the networks

used in distributed systems should be extensible to very large numbers of computers,

whereas individual networks have restricted address spaces and some have performanceSECTION 3.3 NETWORK PRINCIPLES 105

limitations that are incompatible with their large-scale use. In internetworks, a variety

of local and wide area network technologies can be integrated to provide the networking

capacity needed by each group of users. Thus internetworks bring many of the benefits

of open systems to the provision of communication in distributed systems.

Internetworks are constructed from a variety of component networks. They are

interconnected by dedicated switching computers called routers and general-purpose

computers called gateways, and an integrated communication subsystem is produced by

a software layer that supports the addressing and transmission of data to computers

throughout the internetwork. The result can be thought of as a ‘virtual network’

constructed by overlaying an internetwork layer on a communication medium that

consists of the underlying networks, routers and gateways. The Internet is the major

instance of internetworking, and its TCP/IP protocols are an example of this integration

layer.

Network errors • An additional point of comparison not mentioned in Figure 3.1 is the

frequency and types of failure that can be expected in the different types of network. The

reliability of the underlying data transmission media is very high in all types except

wireless networks, where packets are frequently lost due to external interference. But

packets may be lost in all types of network due to processing delays and buffer overflow

at switches and at the destination node. This is by far the most common cause of packet

loss.

Packets may also be delivered in an order different from that in which they were

transmitted. This arises only in networks where separate packets are individually routed

– principally wide area networks. Finally, duplicate copies of packets can be delivered.

This is usually a consequence of an assumption by the sender that a packet has been lost;

the packet is retransmitted, and both the original and the retransmitted copy then turn up

at the destination.

3.3 Network principles

The basis for all computer networks is the packet-switching technique first developed in

the 1960s. This enables data packets addressed to different destinations to share a single

communications link, unlike the circuit-switching technology that underlies conventional telephony. Packets are queued in a buffer and transmitted when the link is available.

Communication is asynchronous – messages arrive at their destination after a delay that

varies depending upon the time that packets take to travel through the network.

3.3.1 Packet transmission

In most applications of computer networks the requirement is for the transmission of

logical units of information, or messages – sequences of data items of arbitrary length.

But before a message is transmitted it is subdivided into packets. The simplest form of

packet is a sequence of binary data (an array of bits or bytes) of restricted length,106 CHAPTER 3 NETWORKING AND INTERNETWORKING

together with addressing information sufficient to identify the source and destination

computers. Packets of restricted length are used:

• so that each computer in the network can allocate sufficient buffer storage to hold

the largest possible incoming packet;

• to avoid the undue delays that would occur in waiting for communication channels

to become free if long messages were transmitted without subdivision.

3.3.2 Data streaming

The transmission and display of audio and video in real time is referred to as streaming.

It requires much higher bandwidths than most other forms of communication in

distributed systems. We have already noted in Chapter 2 that multimedia applications

rely upon the transmission of streams of audio and video data elements at guaranteed

high rates and with bounded latencies.

A video stream requires a bandwidth of about 1.5 Mbps if the data is compressed,

or 120 Mbps if uncompressed. UDP internet packets are generally used to hold the video

frames, but because the flow is continuous as opposed to the intermittent traffic

generated by typical client-server interactions, the packets are handled somewhat

differently. The play time of a multimedia element such as a video frame is the time at

which it must be displayed (for a video element) or converted to sound (for a sound

sample). For example, in a stream of video frames with a frame rate of 24 frames per

second, frame N has a play time that is N/24 seconds after the stream’s start time.

Elements that arrive at their destination later than their play time are no longer useful

and will be dropped by the receiving process.

The timely delivery of audio and video streams depends upon the availability of

connections with adequate quality of service – bandwidth, latency and reliability must

all be considered. Ideally, adequate quality of service should be guaranteed. In general

the Internet does not offer that capability, and the quality of real-time video streams

sometimes reflects that, but in proprietary intranets such as those operated by media

companies, guarantees are sometimes achieved. What is required is the ability to

establish a channel from the source to the destination of a multimedia stream, with a

predefined route through the network, a reserved set of resources at each node through

which it will travel and buffering where appropriate to smooth any irregularities in the

flow of data through the channel. Data can then be passed through the channel from

sender to receiver at the required rate.

ATM networks are specifically designed to provide high bandwidth and low

latencies and to support quality of service by the reservation of network resources. IPv6,

the new network protocol for the Internet outlined in Section 3.4.4, includes features that

enable each of the IP packets in a real-time stream to be identified and treated separately

from other data at the network level.

Communication subsystems that provide quality of service guarantees require

facilities for the preallocation of network resources and the enforcement of the

allocations. The Resource Reservation Protocol (RSVP) [Zhang et al. 1993] enables

applications to negotiate the preallocation of bandwidth for real-time data streams. The

Real Time Transport Protocol (RTP) [Schulzrinne et al. 1996] is an application-level

data transfer protocol that includes details of the play time and other timingSECTION 3.3 NETWORK PRINCIPLES 107

requirements in each packet. The availability of effective implementations of these

protocols in the general Internet will depend upon substantial changes to the transport

and network layers. Chapter 20 discusses the needs of distributed multimedia

applications in more detail.

3.3.3 Switching schemes

A network consists of a set of nodes connected together by circuits. To transmit

information between two arbitrary nodes, a switching system is required. Here we define

the four types of switching that are used in computer networking.

Broadcast • Broadcasting is a transmission technique that involves no switching.

Everything is transmitted to every node, and it is up to potential receivers to notice

transmissions addressed to them. Some LAN technologies, including Ethernet, are

based on broadcasting. Wireless networking is necessarily based on broadcasting, but in

the absence of fixed circuits the broadcasts are arranged to reach nodes grouped in cells.

Circuit switching • At one time telephone networks were the only telecommunication

networks. Their operation was simple to understand: when a caller dialled a number, the

pair of wires from her phone to the local exchange was connected by an automatic

switch at the exchange to the pair of wires connected to the other party’s phone. For a

long-distance call the process was similar but the connection would be switched through

a number of intervening exchanges to its destination. This system is sometimes referred

to as the plain old telephone system, or POTS. It is a typical circuit-switching network.

Packet switching • The advent of computers and digital technology brought many new

possibilities for telecommunication. At the most basic level, it brought processing and

storage. These made it possible to construct a different kind of communication network

called a store-and-forward network. Instead of making and breaking connections to

build circuits, a store-and-forward network just forwards packets from their source to

their destination. There is a computer at each switching node (that is, wherever several

circuits need to be interconnected). Each packet arriving at a node is first stored in

memory at the node and then processed by a program that transmits it on an outgoing

circuit, which transfers the packet to another node that is closer to its ultimate

destination.

There is nothing really new in this idea: the postal system is a store-and-forward

network for letters, with the processing done by humans or machinery at sorting offices.

But in a computer network packets can be stored and processed fast enough to give an

illusion of instantaneous transmission, even though the packet has to be routed through

many nodes.

Frame relay • In reality, it takes anything from a few tens of microseconds to a few

milliseconds to switch a packet through each network node in a store-and-forward

network. This switching delay depends on the packet size, hardware speed and quantity

of other traffic, but its lower bound is determined by the network bandwidth, since the

entire packet must be received before it can be forwarded to another node. Much of the

Internet is based on store-and-forward switching, and as we have already seen, even

short Internet packets typically take up to 200 milliseconds to reach their destinations.

Delays of this magnitude are too long for real-time applications such as telephony and108 CHAPTER 3 NETWORKING AND INTERNETWORKING

video conferencing, where delays of less than 50 milliseconds are needed to sustain

high-quality conversation.

The frame relay switching method brings some of the advantages of circuit

switching to packet-switching networks. They overcome the delay problems by

switching small packets (called frames) on the fly. The switching nodes (which are

usually special-purpose parallel digital processors) route frames based on the

examination of their first few bits; frames as a whole are not stored at nodes but pass

through them as short streams of bits. ATM networks are a prime example; high-speed

ATM networks can transmit packets across networks consisting of many nodes in a few

tens of microseconds.

3.3.4 Protocols

The term protocol is used to refer to a well-known set of rules and formats to be used

for communication between processes in order to perform a given task. The definition

of a protocol has two important parts to it:

• a specification of the sequence of messages that must be exchanged;

• a specification of the format of the data in the messages.

The existence of well-known protocols enables the separate software components of

distributed systems to be developed independently and implemented in different

programming languages on computers that may have different order codes and data

representations.

A protocol is implemented by a pair of software modules located in the sending

and receiving computers. For example, a transport protocol transmits messages of any

length from a sending process to a receiving process. A process wishing to transmit a

message to another process issues a call to a transport protocol module, passing it a

message in the specified format. The transport software then concerns itself with the

transmission of the message to its destination, subdividing it into packets of some

specified size and format that can be transmitted to the destination via the network

protocol – another, lower-level protocol. The corresponding transport protocol module

in the receiving computer receives the packet via the network-level protocol module and

Figure 3.2 Conceptual layering of protocol software

Layer n

Layer 2

Layer 1

Message sent Message received

Communication

medium

Sender Recipient

‚ ‚SECTION 3.3 NETWORK PRINCIPLES 109

performs inverse transformations to regenerate the message before passing it to a

receiving process.

Protocol layers • Network software is arranged in a hierarchy of layers. Each layer

presents an interface to the layers above it that extends the properties of the underlying

communication system. A layer is represented by a module in every computer connected

to the network. Figure 3.2 illustrates the structure and the flow of data when a message

is transmitted using a layered protocol. Each module appears to communicate directly

with a module at the same level in another computer in the network, but in reality data

is not transmitted directly between the protocol modules at each level. Instead, each

layer of network software communicates by local procedure calls with the layers above

and below it.

On the sending side, each layer (except the topmost, or application layer) accepts

items of data in a specified format from the layer above it and applies transformations

to encapsulate the data in the format specified for that layer before passing it to the layer

below for further processing. Figure 3.3 illustrates this process as it applies to the top

four layers of the OSI protocol suite (discussed in the next subsection). The figure shows

the packet headers that hold most network-related data items, but for clarity it omits the

trailers that are present in some types of packet; it also assumes that the application-layer

message to be transmitted is shorter than the underlying network’s maximum packet

size. If not, it would have to be encapsulated in several network-layer packets. On the

receiving side, the converse transformations are applied to data items received from the

layer below before they are passed to the layer above. The protocol type of the layer

above is included in the header of each layer, to enable the protocol stack at the receiver

to select the correct software components to unpack the packets.

Thus each layer provides a service to the layer above it and extends the service

provided by the layer below it. At the bottom is a physical layer. This is implemented

by a communication medium (copper or fibre optic cables, satellite communication

channels or radio transmission) and by analogue signalling circuits that place signals on

the communication medium at the sending node and sense them at the receiving node.

At receiving nodes data items are received and passed upwards through the hierarchy of

software modules, being transformed at each stage until they are in a form that can be

passed to the intended recipient process.

Figure 3.3 Encapsulation as it is applied in layered protocols

Application-layer message

Presentation header

Session header

Transport header

Network header110 CHAPTER 3 NETWORKING AND INTERNETWORKING

Protocol suites • A complete set of protocol layers is referred to as a protocol suite or a

protocol stack, reflecting the layered structure. Figure 3.4 shows a protocol stack that

conforms to the seven-layer Reference Model for Open Systems Interconnection (OSI)

adopted by the International Organization for Standardization (ISO) [ISO 1992]. The

OSI Reference Model was adopted in order to encourage the development of protocol

standards that would meet the requirements of open systems.

The purpose of each level in the OSI Reference Model is summarized in Figure

3.5. As its name implies, it is a framework for the definition of protocols and not a

definition for a specific suite of protocols. Protocol suites that conform to the OSI model

must include at least one specific protocol at each of the seven levels that the model

defines.

Protocol layering brings substantial benefits in simplifying and generalizing the

software interfaces for access to the communication services of networks, but it also

carries significant performance costs. The transmission of an application-level message

via a protocol stack with N layers typically involves N transfers of control to the relevant

layer of software in the protocol suite, at least one of which is an operating system entry,

and taking N copies of the data as a part of the encapsulation mechanism. All of these

overheads result in data transfer rates between application processes that are much lower

than the available network bandwidth.

Figure 3.5 includes examples from protocols used in the Internet, but the

implementation of the Internet does not follow the OSI model in two respects. First, the

application, presentation and session layers are not clearly distinguished in the Internet

protocol stack. Instead, the application and presentation layers are implemented either

as a single middleware layer or separately within each application. Thus CORBA

implements inter-object invocations and data representations in a middleware library

that is included in each application process (see Chapter 8 for further details on

CORBA). Web browsers and other applications that require secure channels employ the

Secure Sockets Layer (Chapter 11) as a procedure library in a similar manner.

Second, the session layer is integrated with the transport layer. Internetwork

protocol suites include an application layer, a transport layer and an internetwork layer.

Figure 3.4 Protocol layers in the ISO Open Systems Interconnection (OSI) protocol model

Application

Presentation

Session

Transport

Network

Data link

Physical

Message sent Message received

Sender Recipient

Layers

Communication

mediumSECTION 3.3 NETWORK PRINCIPLES 111

The internetwork layer is a ‘virtual’ network layer that is responsible for transmitting

internetwork packets to a destination computer. An internetwork packet is the unit of

data transmitted over an internetwork.

Internetwork protocols are overlaid on underlying networks as illustrated in

Figure 3.6. The network interface layer accepts internetwork packets and converts them

into packets suitable for transmission by the network layers of each underlying network.

Packet assembly • The task of dividing messages into packets before transmission and

reassembling them at the receiving computer is usually performed in the transport layer.

The network-layer protocol packets consist of a header and a data field. In most

network technologies, the data field is variable in length, with the maximum length

called the maximum transfer unit (MTU). If the length of a message exceeds the MTU

of the underlying network layer, it must be fragmented into chunks of the appropriate

size, with sequence numbers for use on reassembly, and transmitted in multiple packets.

For example, the MTU for Ethernets is 1500 bytes – no more than that quantity of data

can be transmitted in a single Ethernet packet.

Although the IP protocol stands in the position of a network-layer protocol in the

Internet suite of protocols, its MTU is unusually large at 64 kbytes (8 kbytes is often

used in practice because some nodes are unable to handle such large packets).

Figure 3.5 OSI protocol summary

Layer Description Examples

Application Protocols at this level are designed to meet the communication requirements of

specific applications, often defining the interface to a service.

HTTP, FTP,

SMTP, CORBA

IIOP

Presentation Protocols at this level transmit data in a network representation that is

independent of the representations used in individual computers, which may

differ. Encryption is also performed in this layer, if required.

TLS security,

CORBA data

representation

Session At this level reliability and adaptation measures are performed, such as

detection of failures and automatic recovery.

SIP

Transport This is the lowest level at which messages (rather than packets) are handled.

Messages are addressed to communication ports attached to processes.

Protocols in this layer may be connection-oriented or connectionless.

TCP, UDP

Network Transfers data packets between computers in a specific network. In a WAN or

an internetwork this involves the generation of a route passing through routers.

In a single LAN no routing is required.

IP, ATM virtual

circuits

Data link Responsible for transmission of packets between nodes that are directly

connected by a physical link. In a WAN transmission is between pairs of routers

or between routers and hosts. In a LAN it is between any pair of hosts.

Ethernet MAC,

ATM cell

transfer, PPP

Physical The circuits and hardware that drive the network. It transmits sequences of

binary data by analogue signalling, using amplitude or frequency modulation of

electrical signals (on cable circuits), light signals (on fibre optic circuits) or

other electromagnetic signals (on radio and microwave circuits).

Ethernet baseband signalling,

ISDN112 CHAPTER 3 NETWORKING AND INTERNETWORKING

Whichever MTU value is adopted for IP packets, packets larger than the Ethernet MTU

can arise and they must be fragmented for transmission over Ethernets.

Ports • The transport layer’s task is to provide a network-independent message

transport service between pairs of network ports. Ports are software-defined destination

points at a host computer. They are attached to processes, enabling data transmission to

be addressed to a specific process at a destination node. Next, we discuss the addressing

of ports as they are implemented in the Internet and most other networks. Chapter 4

describes their programming.

Addressing • The transport layer is responsible for delivering messages to destinations

with transport addresses that are composed of the network address of a host computer

and a port number. A network address is a numeric identifier that uniquely identifies a

host computer and enables it to be located by nodes that are responsible for routing data

to it. In the Internet every host computer is assigned an IP number, which identifies it

and the subnet to which it is connected, enabling data to be routed to it from any other

node (as described in the following sections). In Ethernets there are no routing nodes;

each host is responsible for recognizing and picking up packets addressed to it.

Well-known Internet services such as HTTP and FTP have been allocated contact

port numbers and these are registered with a central authority (the Internet Assigned

Numbers Authority (IANA) [www.iana.org I]). To access a service at a given host, a

request is sent to the relevant port at the host. Some services, such as FTP (contact port:

21), then allocate a new port (with a private number) and send the number of the new

port to the client. The client uses the new port for the remainder of a transaction or a

session. Other services, such as HTTP (contact port: 80), transact all of their business

through the contact port.

Port numbers below 1023 are defined as well-known ports whose use is restricted

to privileged processes in most operating systems. The ports between 1024 and 49151

are registered ports for which IANA holds service descriptions, and the remaining ports

up to 65535 are available for private purposes. In practice, all of the ports above 1023

Figure 3.6 Internetwork layers

Underlying network

Application

Network interface

Transport

Internetwork

Internetwork packets

Network-specific packets

Message

Layers

Internetwork

protocols

Underlying

network

protocolsSECTION 3.3 NETWORK PRINCIPLES 113

can be used for private purposes, but computers using them for private purposes cannot

simultaneously access the corresponding registered services.

A fixed port number allocation does not provide an adequate basis for the

development of distributed systems which often include a multiplicity of servers

including dynamically allocated ones. Solutions to this problem involve the dynamic

allocation of ports to services and the provision of binding mechanisms to enable clients

to locate services and their ports using symbolic names. Some of these are discussed

further in Chapter 5.

Packet delivery • There are two approaches to the delivery of packets by the network

layer:

Datagram packet delivery: The term ‘datagram’ refers to the similarity of this

delivery mode to the way in which letters and telegrams are delivered. The essential

feature of datagram networks is that the delivery of each packet is a ‘one-shot’

process; no setup is required, and once the packet is delivered the network retains no

information about it. In a datagram network a sequence of packets transmitted by a

single host to a single destination may follow different routes (if, for example, the

network is capable of adaptation to handle failures or to mitigate the effects of

localized congestion), and when this occurs they may arrive out of sequence.

Every datagram packet contains the full network address of the source and

destination hosts; the latter is an essential parameter for the routing process, which

we describe in the next section. Datagram delivery is the concept on which packet

networks were originally based, and it can be found in most of the computer networks

in use today. The Internet’s network layer (IP), Ethernet and most wired and wireless

local network technologies are based on datagram delivery.

Virtual circuit packet delivery: Some network-level services implement packet

transmission in a manner that is analogous to a telephone network. A virtual circuit

must be set up before packets can pass from a source host A to destination host B.

The establishment of a virtual circuit involves the identification of a route from the

source to the destination, possibly passing through several intermediate nodes. At

each node along the route a table entry is made, indicating which link should be used

for the next stage of the route.

Once a virtual circuit has been set up, it can be used to transmit any number of

packets. Each network-layer packet contains only a virtual circuit number in place of

the source and destination addresses. The addresses are not needed, because packets

are routed at intermediate nodes by reference to the virtual circuit number. When a

packet reaches its destination the source can be determined from the virtual circuit

number.

The analogy with telephone networks should not be taken too literally. In the

POTS a telephone call results in the establishment of a physical circuit from the caller

to the callee, and the voice links from which it is constructed are reserved for their

exclusive use. In virtual circuit packet delivery the circuits are represented only by

table entries in routing nodes, and the links along which the packets are routed are

used only for the time taken to transmit a packet; they are free for other uses for the

rest of the time. A single link may therefore be employed in many separate virtual

circuits. The most important virtual circuit network technology in current use is114 CHAPTER 3 NETWORKING AND INTERNETWORKING

ATM; we have already mentioned (in Section 3.3.3) that it benefits from lower

latencies for the transmission of individual packets; this is a direct result of its use of

virtual circuits. The requirement for a setup phase does, however, result in a short

delay before any packets can be sent to a new destination.

The distinction between datagram and virtual circuit packet delivery in the network

layer should not be confused with a similarly named pair of mechanisms in the transport

layer: connectionless and connection-oriented transmission. We describe these in

Section 3.4.6 in the context of the Internet transport protocols, UDP (connectionless)

and TCP (connection-oriented). Here we simply note that each of these modes of

transmission can be implemented over either type of network layer.

3.3.5 Routing

Routing is a function that is required in all networks except those LANs, such as

Ethernets, that provide direct connections between all pairs of attached hosts. In large

networks, adaptive routing is employed: the best route for communication between two

points in the network is re-evaluated periodically, taking into account the current traffic

in the network and any faults such as broken connections or routers.

The delivery of packets to their destinations in a network such as the one shown

in Figure 3.7 is the collective responsibility of the routers located at connection points.

Unless the source and destination hosts are on the same LAN, the packet has to be

transmitted in a series of hops, passing through router nodes. The determination of

routes for the transmission of packets to their destinations is the responsibility of a

routing algorithm implemented by a program in the network layer at each node.

A routing algorithm has two parts:

1. It must make decisions that determine the route taken by each packet as it travels

through the network. In circuit-switched network layers such as X.25 and framerelay networks such as ATM, the route is determined whenever a virtual circuit or

connection is established. In packet-switched network layers such as IP it is

Figure 3.7 Routing in a wide area network

Hosts

Links

or local

networks

A

D E

B

C

1

2

5

3 4

6

RoutersSECTION 3.3 NETWORK PRINCIPLES 115

determined separately for each packet, and the algorithm must be particularly

simple and efficient if it is not to degrade network performance.

2. It must dynamically update its knowledge of the network based on traffic

monitoring and the detection of configuration changes or failures. This activity is

less time-critical; slower and more computation-intensive techniques can be used.

Both of these activities are distributed throughout the network. The routing decisions are

made on a hop-by-hop basis, using locally held information to determine the next hop

to be taken by each incoming packet. The locally held routing information is updated

periodically by an algorithm that distributes information about the states of the links

(their loads and failure status).

A simple routing algorithm • The algorithm that we describe here is a ‘distance vector’

algorithm. This will provide a basis for the discussion in Section 3.4.3 of the link-state

algorithm that has been used since 1979 as the main routing algorithm in the Internet.

Routing in networks is an instance of the problem of path finding in graphs. Bellman’s

shortest path algorithm, published well before computer networks were developed

[Bellman 1957], provides the basis for the distance vector method. Bellman’s method

was converted into a distributed algorithm suitable for implementation in large networks

by Ford and Fulkerson [1962], and protocols based on their work are often referred to

as ‘Bellman–Ford’ protocols.

Figure 3.8 shows the routing tables that would be held at each of the routers for

the network of Figure 3.7, assuming that the network has no failed links or routers. Each

row provides the routing information for packets addressed to a given destination. The

link field specifies the outgoing link for packets addressed to the destination. The cost

Figure 3.8 Routing tables for the network in Figure 3.7

Routings from A Routings from B Routings from C

To Link Cost To Link Cost To Link Cost

ABCDE

local

1131

01212

ABCDE

1

local

214

10121

ABCDE

22

local

55

21021

Routings from D Routings from E

To Link Cost To Link Cost

ABCDE

336

local

6

12201

ABCDE

4456

local

21110116 CHAPTER 3 NETWORKING AND INTERNETWORKING

field is simply a calculation of the vector distance, or the number of hops to the given

destination. For store-and-forward networks with links of similar bandwidth, this gives

a reasonable estimate of the time for a packet to travel to the destination. The cost

information stored in the routing tables is not used during packet-routing actions taken

by part 1 of the routing algorithm, but it is required for the routing table construction and

maintenance actions in part 2.

The routing tables contain a single entry for each possible destination, showing the

next hop that a packet must take towards its destination. When a packet arrives at a

router the destination address is extracted and looked up in the local routing table. The

resulting entry in the routing table identifies the outgoing link that should be used to

route the packet onwards towards its destination.

For example, when a packet addressed to C is submitted to the router at A, the

router examines the entry for C in its routing table. It shows that the packet should be

routed outwards from A on the link labelled 1. The packet arrives at B and the same

procedure is followed using the routing table at B, which shows that the onward route to

C is via the link labelled 2. When the packet arrives at C the routing table entry shows

‘local’ instead of a link number. This indicates that the packet should be delivered to a

local host.

Now let us consider how the routing tables are built up and how they are

maintained when faults occur in the network – that is, how part 2 of the routing

algorithm described above is performed. Because each routing table specifies only a

single hop for each route, the construction or repair of the routing information can

proceed in a distributed fashion. A router exchanges information about the network with

its neighbouring nodes by sending a summary of its routing table using a router

information protocol (RIP). The RIP actions performed at a router are described

informally as follows:

1. Periodically, and whenever the local routing table changes, send the table (in a

summary form) to all accessible neighbours. That is, send an RIP packet

containing a copy of the table on each non-faulty outgoing link.

2. When a table is received from a neighbouring router, if the received table shows

a route to a new destination, or a better (lower-cost) route to an existing

destination, update the local table with the new route. If the table was received on

link n and it gives a different cost than the local table for a route that begins with

link n, replace the cost in the local table with the new cost. This is done because

the new table was received from a router that is closer to the relevant destination

and is therefore always more authoritative for routes that pass through it.

This algorithm is more precisely described by the pseudo-code program shown in Figure

3.9, where Tr is a table received from another router and Tl is the local table. Ford and

Fulkerson [1962] have shown that the steps described above are sufficient to ensure that

the routing tables will converge on the best routes to each destination whenever there is

a change in the network. The frequency t with which routing tables are propagated, even

when no changes have occurred, is designed to ensure that stability is maintained, for

example, in the case that some RIP packets are lost. The value for t adopted throughout

the Internet is 30 seconds.SECTION 3.3 NETWORK PRINCIPLES 117

To deal with faults, each router monitors its links and acts as follows:

When a faulty link n is detected, set cost to ∞ for all entries in the local table that refer

to the faulty link and perform the Send action.

Thus the information that the link is broken is represented by an infinite value for the

cost to the relevant destinations. When this information is propagated to neighbouring

nodes it will be processed according to the Receive action (note ∞+1 = ∞) and then

propagated further until a node is reached that has a working route to the relevant

destinations, if one exists. The node that still has a working route will eventually

propagate its table, and the working route will replace the faulty one at all nodes.

The vector-distance algorithm can be improved in various ways: costs (also

known as the metric) can be based on the actual bandwidths of the links and the

algorithm can be modified to increase its speed of convergence and to avoid some

undesirable intermediate states, such as loops, that may occur before convergence is

achieved. A routing information protocol with these enhancements was the first routing

protocol used in the Internet, now known as RIP-1 and described in RFC 1058 [Hedrick

1988]. But the solutions for the problems caused by slow convergence are not totally

effective, and this leads to inefficient routing and packet loss while the network is in

intermediate states.

Subsequent developments in routing algorithms have been in the direction of

increasing the amount of knowledge of the network that is held at each node. The most

important family of algorithms of this type are known as link-state algorithms. They are

based on the distribution and updating of a database at each node that represents all, or

a substantial portion, of the network. Each node is then responsible for computing the

optimum routes to the destinations shown in its database. This computation can be

performed by a variety of algorithms, some of which avoid known problems in the

Bellman–Ford algorithm such as slow convergence and undesirable intermediate states.

The design of routing algorithms is a substantial topic, and our discussion of it here is

Figure 3.9 Pseudo-code for RIP routing algorithm

Send: Each t seconds or when Tl changes, send Tl on each non-faulty outgoing link.

Receive: Whenever a routing table Tr is received on link n:

for all rows Rr in Tr {

if (Rr.link  n) {

Rr.cost = Rr.cost + 1;

Rr.link = n;

if (Rr.destination is not in Tl) add Rr to Tl; // add new destination to Tl

else for all rows Rl in Tl {

if (Rr.destination = Rl.destination and

(Rr.cost < Rl.cost or Rl.link = n)) Rl = Rr;

// Rr.cost < Rl.cost : remote node has better route

// Rl.link = n : remote node is more authoritative

}

}

}118 CHAPTER 3 NETWORKING AND INTERNETWORKING

necessarily limited. We return to it in Section 3.4.3 with a description of the operation

of the RIP-1 algorithm, one of the first used for IP routing and still in use in many parts

of the Internet. For extensive coverage of routing in the Internet, see Huitema [2000],

and for further material on routing algorithms in general see Tanenbaum [2003].

3.3.6 Congestion control

The capacity of a network is limited by the performance of its communication links and

switching nodes. When the load at any particular link or node approaches its capacity,

queues will build up at hosts trying to send packets and at intermediate nodes holding

packets whose onward transmission is blocked by other traffic. If the load continues at

the same high level, the queues will continue to grow until they reach the limit of

available buffer space.

Once this state is reached at a node, the node has no option but to drop further

incoming packets. As we have already noted, the occasional loss of packets at the

network level is acceptable and can be remedied by retransmission initiated at higher

levels. But if the rate of packet loss and retransmission reaches a substantial level, the

effect on the throughput of the network can be devastating. It is easy to see why this is

the case: if packets are dropped at intermediate nodes, the network resources that they

have already consumed are wasted and the resulting retransmissions will require a

similar quantity of resources to reach the same point in the network. As a rule of thumb,

when the load on a network exceeds 80% of its capacity, the total throughput tends to

drop as a result of packet losses unless usage of heavily loaded links is controlled.

Instead of allowing packets to travel through the network until they reach overcongested nodes, where they will have to be dropped, it would be better to hold them at

earlier nodes until the congestion is reduced. This will result in increased delays for

packets but will not significantly degrade the total throughput of the network.

Congestion control is the name given to techniques that are designed to achieve this.

In general, congestion control is achieved by informing nodes along a route that

congestion has occurred and that their rate of packet transmission should therefore be

reduced. For intermediate nodes, this will result in the buffering of incoming packets for

a longer period. For hosts that are sources of the packets, the result may be to queue

packets before transmission or to block the application process that is generating them

until the network can handle them.

All datagram-based network layers, including IP and Ethernets, rely on the endto-end control of traffic. That is, the sending node must reduce the rate at which it

transmits packets based only on information that it receives from the receiver.

Congestion information may be supplied to the sending node by explicit transmission of

special messages (called choke packets) requesting a reduction in transmission rate, by

the implementation of a specific transmission control protocol (from which TCP derives

its name – Section 3.4.6 explains the mechanism used in TCP) or by observing the

occurrence of dropped packets (if the protocol is one in which each packet is

acknowledged).

In some virtual circuit based networks, congestion information can be received

and acted on at each node. Although ATM uses virtual circuit delivery, it relies on

quality of service management (see Chapter 20) to ensure that each circuit can carry the

required traffic.SECTION 3.3 NETWORK PRINCIPLES 119

3.3.7 Internetworking

There are many network technologies with different network-, link- and physical-layer

protocols. Local networks are built with Ethernet technologies, while wide area

networks are built over analogue and digital telephone networks of various types,

satellite links and wide area ATM networks. Individual computers and local networks

are linked to the Internet or intranets by modems and by wireless and DSL connections.

To build an integrated network (an internetwork) we must integrate many subnets,

each of which is based on one of these network technologies. To make this possible, the

following are needed:

1. a unified internetwork addressing scheme that enables packets to be addressed to

any host connected to any subnet;

2. a protocol defining the format of internetwork packets and giving rules according

to which they are handled;

3. interconnecting components that route packets to their destinations in terms of

internetwork addresses, transmitting the packets using subnets with a variety of

network technologies.

For the Internet, (1) is provided by IP addresses, (2) is the IP protocol and (3) is

performed by components called Internet routers. The IP protocol and IP addressing are

described in some detail in Section 3.4. Here we describe the functions of Internet

routers and some other components that are used to link networks together.

Figure 3.10 shows a small part of the campus intranet at a British university. Many

of the details shown will be explained in later sections. Here we note that the portion

shown in the figure comprises several subnets interconnected by routers. There are five

subnets, three of which share the IP network 138.37.95 (using the classless interdomain

routing scheme described in Section 3.4.3). The numbers in the diagram are IP

addresses; their structure will be explained in Section 3.4.1. The routers in the diagram

are members of multiple subnets and have an IP address for each subnet, shown against

the connecting links.

The routers (hostnames: hammer and sickle) are, in fact, general-purpose

computers that also fulfil other purposes. One of those purposes is to serve as firewalls;

the role of a firewall is closely linked with the routing function, as we describe in Section

3.4. The 138.37.95.232/29 subnet is not connected to the rest of the network at the IP

level. Only the file server custard can access it to provide a printing service on the

attached printers via a server process that monitors and controls the use of the printers.

All of the links in Figure 3.10 are Ethernets. The bandwidth of most of them is 100

Mbps, but one is 1000 Mbps because it carries a large volume of traffic between a large

number of computers used by students and custard, the file server that holds all of their

files.

There are two Ethernet switches and several Ethernet hubs in the portion of the

network illustrated. Both types of component are transparent to IP packets. An Ethernet

hub is simply a means of connecting together several segments of Ethernet cable, all of

which form a single Ethernet at the network protocol level. All of the Ethernet packets

received by the host are relayed to all of the segments. An Ethernet switch connects120 CHAPTER 3 NETWORKING AND INTERNETWORKING

several Ethernets, routing the incoming packets only to the Ethernet to which the

destination host is connected.

Routers • We have noted that routing is required in all networks except those such as

Ethernets and wireless networks, in which all of the hosts are connected by a single

transmission medium. Figure 3.7 shows such a network with five routers connected by

six links. In an internetwork, the routers may be linked by direct connections, as is

shown in Figure 3.7, or they may be interconnected through subnets, as shown for

custard in Figure 3.10. In both cases, the routers are responsible for forwarding the

internetwork packets that arrive on any connection to the correct outgoing connection,

as explained above. They maintain routing tables for that purpose.

Bridges • Bridges link networks of different types. Some bridges link several networks,

and these are referred to as bridge/routers because they also perform routing functions.

For example, the wider campus network includes a Fibre Distributed Data Interface

Figure 3.10 Simplified view of part of a university campus network

file

compute

dialup

hammer

henry

hotpoint

138.37.88.230

138.37.88.162

bruno

138.37.88.249

router/

sickle

138.37.95.240/29 138.37.95.241

138.37.95.249

138.37.94.247

copper

138.37.88.248

firewall

web

138.37.95.248/29

server

desktop computers 138.37.88.xx

subnet

subnet

Eswitch

138.37.88

server

server

server

138.37.88.251

custard

138.37.94.246

desktop computers 138.37.94.xx

Eswitch

138.37.94

hub hub

Staff subnet Student subnet

other

servers

router/

firewall

138.37.94.251

138.37.88.247

file server/

gateway

printers

138.37.95.232/29

subnet

Campus

router

Campus

router Subnet boundary

100 Mbps Ethernet

1000 Mbps Ethernet

Eswitch Ethernet switchSECTION 3.3 NETWORK PRINCIPLES 121

(FDDI) backbone (not shown on Figure 3.10), and this is linked to the Ethernet subnets

in the figure by bridge/routers.

Hubs • Hubs are simply a convenient means of connecting hosts and extending

segments of Ethernet and other broadcast local network technologies. They have a

number of sockets (typically 4–64), to each of which a host computer can be connected.

They can also be used to overcome the distance limitations on single segments and

provide a means of adding additional hosts.

Switches • Switches perform a similar function to routers, but for local networks

(normally Ethernets) only. That is, they interconnect several separate Ethernets, routing

the incoming packets to the appropriate outgoing network. They perform their task at the

level of the Ethernet network protocol. When they start up they have no knowledge of

the wider internetwork and build up routing tables by the observation of traffic,

supplemented by broadcast requests when they lack information.

The advantage of switches over hubs is that they separate the incoming traffic and

transmit it only on the relevant outgoing network, reducing congestion on the other

networks to which they are connected.

Tunnelling • Bridges and routers transmit internetwork packets over a variety of

underlying networks by translating between their network-layer protocols and an

internetwork protocol, but there is one situation in which the underlying network

protocol can be hidden from the layers above it without the use of an internetwork

protocol. A pair of nodes connected to separate networks of the same type can

communicate through another type of network by constructing a protocol ‘tunnel’. A

protocol tunnel is a software layer that transmits packets through an alien network

environment.

The following analogy explains the reason for the choice of terminology and

provides another way to think about tunnelling. A tunnel through a mountain enables a

road to transport cars where it would otherwise be impossible. The road is continuous –

the tunnel is transparent to the application (cars). The road is the transport mechanism,

and the tunnel enables it to work in an alien environment.

Figure 3.11 illustrates the use of tunnelling to support the migration of the Internet

to the IPv6 protocol. IPv6 is intended to replace the version of IP still widely in use,

IPv4, and is incompatible with it. (Both IPv4 and IPv6 are described in Section 3.4.)

During the period of transition to IPv6 there will be ‘islands’ of IPv6 networking in the

sea of IPv4. In our illustration A and B are such islands. At the boundaries of islands

Figure 3.11 Tunnelling for IPv6 migration

A IPv6 IPv6 B

IPv6 encapsulated in IPv4 packets

Encapsulators

IPv4 network122 CHAPTER 3 NETWORKING AND INTERNETWORKING

IPv6 packets are encapsulated in IPv4 and transported over the intervening IPv4

networks in that manner.

For another example, MobileIP (described in Section 3.4.5) transmits IP packets

to mobile hosts anywhere in the Internet by constructing a tunnel to them from their

home base. The intervening network nodes do not need to be modified to handle the

MobileIP protocol. The IP multicast protocol is handled in a similar way, relying on a

few routers that support IP multicast routing to determine the routes, but transmitting IP

packets through other routers using standard IP addresses. The PPP protocol for the

transmission of IP packets over serial links provides yet another example.

3.4 Internet protocols

We describe here the main features of the TCP/IP suite of protocols and discuss their

advantages and limitations when used in distributed systems.

The Internet emerged from two decades of research and development work on

wide area networking in the USA, commencing in the early 1970s with the ARPANET

– the first large-scale computer network development [Leiner et al. 1997]. An important

part of that research was the development of the TCP/IP protocol suite. TCP stands for

Transmission Control Protocol, IP for Internet Protocol. The widespread adoption of the

TCP/IP and Internet application protocols in national research networks, and more

recently in commercial networks in many countries, has enabled the national networks

to be integrated into a single internetwork that has grown extremely rapidly to its present

size, with more than 60 million hosts. Many application services and application-level

protocols (shown in parentheses in the following list) now exist based on TCP/IP,

including the Web (HTTP), email (SMTP, POP), netnews (NNTP), file transfer (FTP)

and Telnet (telnet). TCP is a transport protocol; it can be used to support applications

directly, or additional protocols can be layered on it to provide additional features. For

Figure 3.12 TCP/IP layers

Messages (UDP) or streams (TCP)

Application

Transport

Internet

UDP or TCP packets

IP datagrams

Network-specific frames

Message

Layers

Underlying network

Network interfaceSECTION 3.4 INTERNET PROTOCOLS 123

example, HTTP is usually transported by the direct use of TCP, but when end-to-end

security is required, the Transport Layer Security (TLS) protocol (described in Section

11.6.3) is layered on top of TCP to produce secure channels and HTTP messages are

transmitted via the secure channels.

The Internet protocols were originally developed primarily to support simple wide

area applications such as file transfer and electronic mail, involving communication

with relatively high latencies between geographically dispersed computers, but they

turned out to be efficient enough to support the requirements of many distributed

applications on both wide area and local networks and they are now almost universally

used in distributed systems. The resulting standardization of communication protocols

has brought immense benefits.

The general illustration of internetwork protocol layers of Figure 3.6 is translated

into the specific Internet case in Figure 3.12. There are two transport protocols – TCP

(Transport Control Protocol) and UDP (User Datagram Protocol). TCP is a reliable

connection-oriented protocol, and UDP is a datagram protocol that does not guarantee

reliable transmission. The Internet Protocol is the underlying ‘network’ protocol of the

Internet virtual network – that is, IP datagrams provide the basic transmission

mechanism for the Internet and other TCP/IP networks. We placed the word ‘network’

in quotation marks in the preceding sentence because it is not the only network layer

involved in the implementation of Internet communication. This is because the Internet

protocols are usually layered over another network technology, such as Ethernet, which

already provides a network layer that enables the computers attached to the same

network to exchange datagrams. Figure 3.13 illustrates the encapsulation of packets that

would occur for the transmission of a message via TCP over an underlying Ethernet. The

tags in the headers are the protocol types for the layers above, needed for the receiving

protocol stack to correctly unpack the packets. In the TCP layer, the receiver’s port

number serves a similar purpose, enabling the TCP software component at the receiving

host to pass the message to a specific application-level process.

The TCP/IP specifications [Postel 1981a; 1981b] do not specify the layers below

the Internet datagram layer – IP packets in the Internet layer are transformed into packets

for transmission over almost any combination of underlying networks or data links.

For example, IP ran initially over the ARPANET, which consisted of hosts and an

early version of routers (called PSEs) connected by long-distance data links. Today it is

Figure 3.13 Encapsulation as it occurs when a message is transmitted via TCP over an Ethernet

Application message

TCP header

Ethernet header

Ethernet frame

port

TCP

IP

IP header124 CHAPTER 3 NETWORKING AND INTERNETWORKING

used over virtually every known network technology, including ATM, local area

networks such as Ethernets, and token ring networks. IP is implemented over serial lines

and telephone circuits via the PPP protocol [Parker 1992], enabling it to be used for

communication with modem connections and other serial links.

The success of TCP/IP is based on the protocols’ independence from the

underlying transmission technology, enabling internetworks to be built up from many

heterogeneous networks and data links. Users and application programs perceive a

single virtual network supporting TCP and UDP and implementors of TCP and UDP see

a single virtual IP network, hiding the diversity of the underlying transmission media.

Figure 3.14 illustrates this view.

In the next two sections we describe the IP addressing scheme and the IP protocol.

The Domain Name System – which converts domain names such as www.amazon.com,

hpl.hp.com, stanford.edu and qmw.ac.uk, with which Internet users are so familiar, into

IP addresses – is introduced in Section 3.4.7 and described more fully in Chapter 13.

The version of IP in predominant use throughout the Internet is IPv4 (since

January 1984), and that is the version that we shall describe in the next two sections. But

the rapid growth in the use of the Internet led to the publication of a specification of a

new version (IPv6) to overcome the addressing limitations of IPv4 and add features to

support some new requirements. We describe IPv6 in Section 3.4.4. Because of the vast

amount of software that will be affected, a gradual migration to IPv6 is planned over a

period of 10 years or more.

3.4.1 IP addressing

Perhaps the most challenging aspect of the design of the Internet protocols was the

construction of schemes for naming and addressing hosts and for routing IP packets to

their destinations. The scheme used for assigning host addresses to networks and the

computers connected to them had to satisfy the following requirements:

• It must be universal – any host must be able to send packets to any other host in

the Internet.

• It must be efficient in its use of the address space – it is impossible to predict the

ultimate size of the Internet and the number of network and host addresses likely

to be required. The address space must be carefully partitioned to ensure that

addresses will not run out. In 1978–82, when the specifications for the TCP/IP

protocols were being developed, provision for 232 or approximately 4 billion

addressable hosts (about the same as the population of the world at that time) was

Figure 3.14 The programmer's conceptual view of a TCP/IP Internet

IP

Application Application

TCP UDPSECTION 3.4 INTERNET PROTOCOLS 125

considered adequate. This judgement has proved to be short-sighted, for two

reasons:

– The rate of growth of the Internet has far outstripped all predictions.

– The address space has been allocated and used much less efficiently than

expected.

• The addressing scheme must lend itself to the development of a flexible and

efficient routing scheme, but the addresses themselves cannot contain very much

of the information needed to route a packet to its destination.

Today the overwhelming majority of Internet traffic continues to use the IP version 4

address and packet format defined three decades ago. The scheme assigns an IP address

to each host in the Internet – a 32-bit numeric identifier containing a network identifier,

which uniquely identifies one of the subnetworks in the Internet, and a host identifier,

which uniquely identifies the host’s connection to that network. It is these addresses that

are placed in IP packets and used to route them to their destinations.

The design adopted for the Internet address space is shown in Figure 3.15. There

are four allocated classes of Internet address – A, B, C and D. Class D is reserved for

Internet multicast communication, which is implemented in only some Internet routers

and is discussed further in Section 4.4.1. Class E contains a range of unallocated

addresses, which are reserved for future requirements.

These 32-bit Internet addresses, containing a network identifier and host

identifier, are usually written as a sequence of four decimal numbers separated by dots.

Each decimal number represents one of the four bytes, or octets, of the IP address. The

permissible values for each class of network address are shown in Figure 3.16.

Three classes of address were designed to meet the requirements of different types

of organization. The Class A addresses, with a capacity for 224 hosts on each subnet, are

reserved for very large networks such as the US NSFNet and other national wide area

networks. Class B addresses are allocated to organizations that operate networks likely

Figure 3.15 Internet address structure, showing field sizes in bits

7 24

Class A: 0 Network ID Host ID

14 16

Class B: 1 0 Network ID Host ID

21 8

Class C: 1 1 0 Network ID Host ID

28

Class D (multicast): 1 1 1 0 Multicast address

Class E (reserved): 1 1 1 1 unused

28126 CHAPTER 3 NETWORKING AND INTERNETWORKING

to contain more than 255 computers, and Class C addresses are allocated to all other

network operators.

Internet addresses with host identifiers 0 and all 1s (binary) are used for special

purposes. Addresses with the host identifier set to 0 are used to refer to ‘this host’, and

a host identifier that is all 1s is used to address a broadcast message to all of the hosts

connected to the network specified in the network identifier part of the address.

Network identifiers are allocated by the Internet Assigned Numbers Authority

(IANA) to organizations with networks connected to the Internet. Host identifiers for the

computers on each network connected to the Internet are assigned by the managers of

the relevant networks.

Since host addresses include a network identifier, any computer that is connected

to more than one network must have separate addresses on each, and whenever a

computer is moved to a different network, its Internet address must change. These

requirements can lead to substantial administrative overheads, for example in the case

of portable computers.

In practice, the IP address allocation scheme has not turned out to be very

effective. The main difficulty is that network administrators in user organizations cannot

easily predict future growth in their need for host addresses, and they tend to

overestimate, requesting Class B addresses when in doubt. Around 1990 it became

evident that based on the rate of allocation at the time, IP addresses were likely to run

out around 1996. Three steps were taken. The first was to initiate the development of a

new IP protocol and addressing scheme, the result of which was the specification of

IPv6.

The second step was to radically modify the way in which IP addresses were

allocated. A new address allocation and routing scheme designed to make more

effective use of the IP address space was introduced, called classless interdomain

routing (CIDR). We describe CIDR in Section 3.4.3. The local network illustrated in

Figure 3.16 Decimal representation of Internet addresses

octet 1 octet 2 octet 3

Class A: 1 to 127

0 to 255 0 to 255 1 to 254

Class B: 128 to 191

Class C: 192 to 223

Class D (multicast): 224 to 239

Network ID

Network ID

Network ID

Host ID

Host ID

Host ID

Multicast address

0 to 255 0 to 255 1 to 254

0 to 255 0 to 255 0 to 255

0 to 255 0 to 255 0 to 255

Class E (reserved): 240 to 255 0 to 255 0 to 255 1 to 254

1.0.0.0 to

127.255.255.255

128.0.0.0 to

191.255.255.255

192.0.0.0 to

223.255.255.255

224.0.0.0 to

239.255.255.255

240.0.0.0 to

255.255.255.255

Range of addressesSECTION 3.4 INTERNET PROTOCOLS 127

Figure 3.10 includes several Class C sized subnets in the range 138.37.88–138.37.95,

linked by routers. The routers manage the delivery of IP packets to all of the subnets.

They also handle traffic between the subnets and from the subnets to the rest of the

world. The figure also illustrates the use of CIDR to subdivide a Class B address space

to produce several Class C sized subnets.

The third step was to enable unregistered computers to access the Internet

indirectly through routers that implement a Network Address Translation (NAT)

scheme. We describe this scheme in Section 3.4.3.

3.4.2 The IP protocol

The IP protocol transmits datagrams from one host to another, if necessary via

intermediate routers. The full IP packet format is rather complex, but Figure 3.17 shows

the main components. There are several header fields, not shown in the diagram, that are

used by the transmission and routing algorithms.

IP provides a delivery service that is described as offering unreliable or best-effort

delivery semantics, because there is no guarantee of delivery. Packets can be lost,

duplicated, delayed or delivered out of order, but these errors arise only when the

underlying networks fail or buffers at the destination are full. The only checksum in IP

is a header checksum, which is inexpensive to calculate and ensures that any corruptions

in the addressing and packet management data will be detected. There is no data

checksum, which avoids overheads when crossing routers, leaving the higher-level

protocols (TCP and UDP) to provide their own checksums – a practical instance of the

end-to-end argument (Section 2.3.3).

The IP layer puts IP datagrams into network packets suitable for transmission in

the underlying network (which might, for example, be an Ethernet). When an IP

datagram is longer than the MTU of the underlying network, it is broken into smaller

packets at the source and reassembled at its final destination. Packets can be further

broken up to suit the underlying networks encountered during the journey from source

to destination. (Each packet has a fragment identifier to enable out-of-order fragments

to be collected.)

The IP layer must also insert a ‘physical’ network address of the message

destination to the underlying network. It obtains this from the address resolution module

in the Internet network interface layer, which is described in the next subsection.

Address resolution • The address resolution module is responsible for converting

Internet addresses to network addresses for a specific underlying network (sometimes

called physical addresses). For example, if the underlying network is an Ethernet, the

address resolution module converts 32-bit Internet addresses to 48-bit Ethernet

addresses.

Figure 3.17 IP packet layout

IP address of source IP address of destination data

header

up to 64 kilobytes128 CHAPTER 3 NETWORKING AND INTERNETWORKING

This translation is network technology dependent:

• Some hosts are connected directly to Internet packet switches; IP packets can be

routed to them without address translation.

• Some local area networks allow network addresses to be assigned to hosts

dynamically, and the addresses can be conveniently chosen to match the host

identifier portion of the Internet address – translation is simply a matter of

extracting the host identifier from the IP address.

• For Ethernets and some other local networks, the network address of each

computer is hard-wired into its network interface hardware and bears no direct

relation to its Internet address – translation depends upon knowledge of the

correspondence between IP addresses and addresses for the hosts on the local

network and is done using an address resolution protocol (ARP).

We now outline the implementation of an ARP for Ethernets. It uses dynamic enquiries

in order to operate correctly when computers are added to a local network but exploits

caching to minimize enquiry messages. Consider first the case in which a host computer

connected to an Ethernet uses IP to transmit a message to another computer on the same

Ethernet. The IP software module on the sending computer must translate the recipient’s

Internet address that it finds in the IP packet to an Ethernet address before the packet can

be delivered. It invokes the ARP module on the sending computer to do so.

The ARP module on each host maintains a cache of (IP address, Ethernet address)

pairs that it has previously obtained. If the required IP address is in the cache, then the

query is answered immediately. If not, then ARP transmits an Ethernet broadcast packet

(an ARP request packet) on the local Ethernet containing the desired IP address. Each

of the computers on the local Ethernet receives the ARP request packet and checks the

IP address in it to see whether it matches its own IP address. If it does, an ARP reply

packet is sent to the originator of the ARP request containing the sender’s Ethernet

address; otherwise the ARP request packet is ignored. The originating ARP module adds

the new IP address to Ethernet address mapping to its local cache of (IP address,

Ethernet address) pairs so that it can respond to similar requests in the future without

broadcasting an ARP request. Eventually, the ARP cache at each computer will contain

an (IP address, Ethernet address) pair for all of the computers that IP packets are sent

to. Thus ARP broadcasts will be needed only when a computer is newly connected to

the local Ethernet.

IP spoofing • We have seen that IP packets include a source address – the IP address of

the sending computer. This, together with a port address encapsulated in the data field

(for UDP and TCP packets), is often used by servers to generate a return address.

Unfortunately, it is not possible to guarantee that the source address given is in fact the

address of the sender. A malicious sender can easily substitute an address that is

different from its own. This loophole has been the source of several well-known attacks,

including the distributed denial of service attacks of February 2000 [Farrow 2000]

mentioned in Chapter 1, Section 1.5.3. The method used was to issue many ping service

requests to a large number of computers at several sites (ping is a simple service

designed to check the availability of a host). These malicious ping requests all contained

the IP address of a target computer in their sender address field. The ping responses wereSECTION 3.4 INTERNET PROTOCOLS 129

therefore all directed to the target, whose input buffers were overwhelmed, preventing

any legitimate IP packets getting through. This attack is discussed further in Chapter 11.

3.4.3 IP routing

The IP layer routes packets from their source to their destination. Each router in the

Internet implements IP-layer software to provide a routing algorithm.

Backbones • The topological map of the Internet is partitioned conceptually into

autonomous systems (ASs), which are subdivided into areas. The intranets of most large

organizations such as universities and large companies are regarded as ASs, and they

will usually include several areas. In Figure 3.10, the campus intranet is an AS and the

portion shown is an area. Every AS in the topological map has a backbone area. The

collection of routers that connect non-backbone areas to the backbone and the links that

interconnect those routers are called the backbone of the network. The links in the

backbone are usually of high bandwidth and are replicated for reliability. This hierarchic

structure is a conceptual one that is exploited primarily for the management of resources

and the maintenance of the components. It does not affect the routing of IP packets.

Routing protocols • RIP-1, the first routing algorithm used in the Internet, is a version

of the distance-vector algorithm described in Section 3.3.5. RIP-2 (described in RFC

1388 [Malkin 1993]) was developed from it to accommodate several additional

requirements, including classless interdomain routing, better multicast routing and the

need for authentication of RIP packets to prevent attacks on the routers.

As the scale of the Internet has expanded and the processing capacity of routers

has increased, there has been a move towards the adoption of algorithms that do not

suffer from the slow convergence and potential instability of distance-vector algorithms.

The direction of the move is towards the link-state class of algorithms mentioned in

Section 3.3.5 and the algorithm called open shortest path first (OSPF). This protocol is

based on a path-finding algorithm that is due to Dijkstra [1959] and has been shown to

converge more rapidly than the RIP algorithm.

We should note that the adoption of new routing algorithms in IP routers can

proceed incrementally. A change in routing algorithm results in a new version of the RIP

protocol, and a version number is carried by each RIP packet. The IP protocol does not

change when a new RIP protocol is introduced. Any IP router will correctly forward

incoming IP packets on a reasonable, if not optimum, route, whatever version of RIP

they use. But for routers to cooperate in the updating of their routing tables, they must

share a similar algorithm. For this purpose the topological areas defined above are used.

Within each area a single routing algorithm applies, and the routers within an area

cooperate in the maintenance of their routing tables. Routers that support only RIP-1 are

still commonplace and they coexist with routers that support RIP-2 and OSPF, using

backwards-compatibility features incorporated in the newer protocols.

In 1993, empirical observations [Floyd and Jacobson 1993] showed that the 30-

second frequency with which RIP routers exchange information was producing a

periodicity in the performance of IP transmissions. The average latency for IP packet

transmissions showed a peak at 30-second intervals. This was traced to the behaviour of

routers performing the RIP protocol – on receipt of an RIP packet, routers would delay

the onward transmission of any IP packets that they held until the routing table update130 CHAPTER 3 NETWORKING AND INTERNETWORKING

process was complete for all RIP packets received to date. This tended to cause the

routers to perform the RIP actions in lock-step. The correction recommended was for

routers to adopt a random value in the range of 15–45 seconds for the RIP update period.

Default routes • Up to now, our discussion of routing algorithms has suggested that

every router maintains a full routing table showing the route to every destination (subnet

or directly connected host) in the Internet. At the current scale of the Internet this is

clearly infeasible (the number of destinations is probably already in excess of 1 million

and still growing very rapidly).

Two possible solutions to this problem come to mind, and both have been adopted

in an effort to alleviate the effects of the Internet’s growth. The first solution is to adopt

some form of topological grouping of IP addresses. Prior to 1993, nothing could be

inferred from an IP address about its location. In 1993, as part of the move to simplify

and economize on the allocation of IP addresses that is discussed below under CIDR,

the decision was taken that for future allocations, the following regional locations would

be applied:

Addresses 194.0.0.0 to 195.255.255.255 are in Europe

Addresses 198.0.0.0 to 199.255.255.255 are in North America

Addresses 200.0.0.0 to 201.255.255.255 are in Central and South America

Addresses 202.0.0.0 to 203.255.255.255 are in Asia and the Pacific

Because these geographical regions also correspond to well-defined topological regions

in the Internet and just a few gateway routers provide access to each region, this enables

a substantial simplification of routing tables for those address ranges. For example, a

router outside Europe can have a single table entry for the range of addresses 194.0.0.0

to 195.255.255.255 that sends all IP packets with destinations in that range on the same

route to the nearest European gateway router. But note that before the date of that

decision, IP addresses were allocated largely without regard to topology or geography.

Many of those addresses are still in use, and the 1993 decision does nothing to reduce

the scale of routing table entries for those addresses.

The second solution to the routing table size explosion probem is simpler and very

effective. It is based on the observation that the accuracy of routing information can be

relaxed for most routers as long as some key routers (those closest to the backbone links)

have relatively complete routing tables. The relaxation takes the form of a default

destination entry in routing tables. The default entry specifies a route to be used for all

IP packets whose destinations are not included in the routing table. To illustrate this,

consider Figures 3.7 and 3.8 and suppose that the routing table for node C is altered to

show:

Routings from C

To Link Cost

BCE

Default

2

local

55

101-SECTION 3.4 INTERNET PROTOCOLS 131

Thus node C is ignorant of nodes A and D. It will route all packets addressed to them

via link 5 to E. What is the consequence? Packets addressed to D will reach their

destination without loss of efficiency in routing, but packets addressed to A will make

an extra hop, passing through E and B on the way. In general, the use of default routings

trades routing efficiency for table size. But in some cases, especially where a router is

on a spur, so that all outward messages must pass through a single point, there is no loss

of efficiency. The default routing scheme is heavily used in Internet routing; no single

router holds routes to all destinations in the Internet.

Routing on a local subnet • Packets addressed to hosts on the same network as the

sender are transmitted to the destination host in a single hop, using the host identifier

part of the address to obtain the address of the destination host on the underlying

network. The IP layer simply uses ARP to get the network address of the destination and

then uses the underlying network to transmit the packets.

If the IP layer in the sending computer discovers that the destination is on a

different network, it must send the message to a local router. It uses ARP to get the

network address of the gateway or router and then uses the underlying network to

transmit the packet to it. Gateways and routers are connected to two or more networks

and they have several Internet addresses, one for each network to which they are

attached.

Classless interdomain routing (CIDR) • The shortage of IP addresses referred to in

Section 3.4.1 led to the introduction in 1996 of this scheme for allocating addresses and

managing the entries in routing tables. The main problem was a scarcity of Class B

addresses – those for subnets with more than 255 hosts connected. Plenty of Class C

addresses were available. The CIDR solution for this problem is to allocate a batch of

contiguous Class C addresses to a subnet requiring more than 255 addresses. The CIDR

scheme also makes it possible to subdivide a Class B address space for allocation to

multiple subnets.

Batching Class C addresses sounds like a straightforward step, but unless it is

accompanied by a change in routing table format, it has a substantial impact on the size

of routing tables and hence the efficiency of the algorithms that manage them. The

change adopted was to add a mask field to the routing tables. The mask is a bit pattern

that is used to select the portion of an IP address that is compared with the routing table

entry. This effectively enables the host/subnet address to be any portion of the IP

address, providing more flexibility than the classes A, B and C – hence the name

classless interdomain routing. Once again, these changes to routers are made on an

incremental basis, so some routers perform CIDR and others use the old class-based

algorithms.

This works because the newly allocated ranges of Class C addresses are assigned

modulo 256, so each range represents an integral number of Class C sized subnet

addresses. On the other hand, some subnets also make use of CIDR to subdivide the

range of addresses in a single network, of Class A, B or C. If a collection of subnets is

connected to the rest of the world entirely by CIDR routers, then the ranges of IP

addresses used within the collection can be allocated to individual subnets in chunks

determined by a binary mask of any size.

For example, a Class C address space can be subdivided into 32 groups of 8.

Figure 3.10 contains an example of the use of the CIDR mechanism to split the132 CHAPTER 3 NETWORKING AND INTERNETWORKING

138.37.95 Class C sized subnet into several groups of eight host addresses that are

routed differently. The separate groups are denoted by notations 138.37.95.232/29,

138.37.95.248/29 and so on. The /29 portion of these addresses denotes an attached 32-

bit binary mask with 29 leading 1s and three trailing 0s.

Unregistered addresses and Network Address Translation (NAT) • Not all of the

computers and devices that access the Internet need to be assigned globally unique IP

addresses. Computers that are attached to a local network and access to the Internet

through a NAT-enabled router can rely upon the router to redirect incoming UDP and

TCP packets for them. Figure 3.18 illustrates a typical home network with computers

and other network devices linked to the Internet through a NAT-enabled router. The

network includes Internet-enabled computers that are connected to the router by a wired

Ethernet connection as well as others that are connected through a WiFi access point.

For completeness some Bluetooth-enabled devices are shown, but these are not

connected to the router and hence cannot access the Internet directly. The home network

has been allocated a single registered IP address (83.215.152.95) by its Internet service

provider. The approach described here is suitable for any organization wishing to

connect computers without registered IP addresses to the Internet.

All of the Internet-enabled devices on the home network have been assigned

unregistered IP addresses on the 192.168.1.x Class C subnet. Most of the internal

computers and devices are allocated individual IP addresses dynamically by a Dynamic

Figure 3.18 A typical NAT-based home network

83.215.152.95

Ethernet switch

Modem / firewall / router (NAT-enabled)

printer

DSL or cable

connection to ISP

192.168.1.xx subnet

PC 1

WiFi base station/

access point 192.168.1.10

192.168.1.5

192.168.1.2

192.168.1.1

192.168.1.104 PC 2

192.168.1.101

Laptop

192.168.1.105

Game box

192.168.1.106

Media hub

TV monitor

Bluetooth

adapter

Bluetooth

printer

CameraSECTION 3.4 INTERNET PROTOCOLS 133

Host Configuration Protocol (DHCP) service running on the router. In our illustration

the numbers above 192.168.1.100 are used by the DHCP service and the nodes with

lower numbers (such as PC 1) have been allocated numbers manually, for a reason

explained later in this subsection. Although all of these addresses are completely hidden

from the rest of the Internet by the NAT router, it is conventional to use a range of

addresses from one of three blocks of addresses (10.z.y.x, 172.16.y.x or 192.168.y.x)

that IANA has reserved for private internets.

NAT is described in RFC 1631 [Egevang and Francis 1994] and extended in RFC

2663 [Srisuresh and Holdrege 1999]. NAT-enabled routers maintain an address

translation table and exploit the source and destination port number fields in the UDP

and TCP packets to assign each incoming reply message to the internal computer that

sent the corresponding request message. Note that the source port given in a request

message is always used as the destination port in the corresponding reply message.

The most commonly used variant of NAT addressing works as follows:

– When a computer on the internal network sends a UDP or TCP packet to a

computer outside it, the router receives the packet and saves the source IP

address and port number to an available slot in its address translation table.

– The router replaces the source address in the packet with the router’s IP address

and the source port with a virtual port number that indexes the table slot

containing the sending computer’s address information.

– The packet with the modified source address and port number is then

forwarded towards its destination by the router. The address translation table

now holds a mapping from virtual port numbers to real internal IP addresses

and port numbers for all packets sent recently by computers on the internal

network.

– When the router receives a UDP or TCP packet from an external computer it

uses the destination port number in the packet to access a slot in the address

translation table. It replaces the destination address and destination port in the

received packet with those stored in the slot and forwards the modified packet

to the internal computer identified by the destination address.

The router will retain a port mapping and reuse it as long as it appears to be in use. A

timer is reset each time the router accesses an entry in the table. If the entry is not

accessed again before the timer expires, the entry is removed from the table.

The scheme described above deals satisfactorily with the commonest modes of

communication for nonregistered computers, in which they act as clients to external

services such as web servers. But it does not enable them to act as servers to handle

incoming requests. To deal with that case, NAT routers can be configured manually to

forward all of the incoming requests on a given port to one particular internal computer.

Computers that act as servers must retain the same internal IP address and this is

achieved by allocating their addresses manually (as was done for PC 1). This solution to

the problem of providing external access to services is satisfactory as long as there is no

requirement for more than one internal computer to offer a service on any given port.

NAT was introduced as a short-term solution to the problem of IP address

allocation for personal and home computers. Its has enabled the expansion of Internet

use to proceed far further than was originally anticipated, but it does impose some134 CHAPTER 3 NETWORKING AND INTERNETWORKING

limitations, of which the last point is an example. IPv6 must be seen as the next step,

enabling full Internet participation for all computers and portable devices.

3.4.4 IP version 6

A more permanent solution to the addressing limitations of IPv4 was also pursued, and

this led to the development and adoption of a new version of the IP protocol with

substantially larger addresses. The IETF noticed the potential problems arising from the

32-bit addresses of IPv4 as early as 1990 and initiated a project to develop a new version

of the IP protocol. IPv6 was adopted by the IETF in 1994 and a strategy for migration

to it was recommended.

Figure 3.19 shows the layout of IPv6 headers. We do not propose to cover their

construction in detail here. Readers are referred to Tanenbaum [2003] or Stallings

[2002] for tutorial material on IPv6 and to Huitema [1998] for a blow-by-blow account

of the IPv6 design process and implementation plans. Here we will outline the main

advances that IPv6 embodies:

Address space: IPv6 addresses are 128 bits (16 bytes) long. This provides for a truly

astronomical number of addressable entities: 2128, or approximately 3 × 1038.

Tanenbaum calculates that this is sufficient to provide 7 × 1023 IP addresses per

square metre across the entire surface of the Earth. More conservatively, Huitema

made a calculation assuming that IP addresses are allocated as inefficiently as

telephone numbers and came up with a figure of 1000 IP addresses per square metre

of the Earth’s surface (land and water).

The IPv6 address space is partitioned. We cannot detail the partitioning here,

but even the minor partitions (one of which will hold the entire range of IPv4

addresses, mapped one-to-one) are far larger than the total IPv4 space. Many

partitions (representing 72% of the total) are reserved for purposes as yet undefined.

Two large partitions (each comprising 1/8 of the address space) are allocated for

general purposes and will be assigned to normal network nodes. One of them is

intended to be organized according to the geographic locations of the addressed

nodes and the other according to their organizational locations. This allows two

Figure 3.19 IPv6 header layout

Source address

(128 bits)

Destination address

(128 bits)

Version (4 bits) Traffic class (8 bits) Flow label (20 bits)

Payload length (16 bits) Hop limit (8 bits) Next header (8 bits)SECTION 3.4 INTERNET PROTOCOLS 135

alternative strategies for aggregating addresses for routing purposes – it remains to

be seen which will prove more effective or popular.

Routing speed: The complexity of the basic IPv6 header and the processing required

at each node are reduced. No checksum is applied to the packet content (payload),

and no fragmentation can occur once a packet has begun its journey. The former is

considered acceptable because errors can be detected at higher levels (TCP does

include a content checksum), and the latter is achieved by supporting a mechanism

for determining the smallest MTU before a packet is transmitted.

Real-time and other special services: The traffic class and flow label fields are

concerned with this. Multimedia streams and other sequences of real-time data

elements can be transmitted as part of an identified flow. The first 6 bits of the traffic

class field can be used with the flow label or independently to enable specific packets

to be handled more rapidly or with higher reliability than others. Traffic class values

0 through 8 are for transmissions that can be slowed without disastrous effects on the

application. Other values are reserved for packets whose delivery is time-dependent.

Such packets must either be delivered promptly or dropped – late delivery is of no

value.

Flow labels enable resources to be reserved in order to meet the timing

requirements of specific real-time data streams, such as live audio and video

transmissions. Chapter 20 discusses these requirements and methods for the

allocation of resources for them. Of course, the routers and transmission links in the

Internet have limited resources, and the concept of reserving them for specific users

and applications has not previously been considered. The use of these facilities of

IPv6 will depend upon major enhancements to the infrastructure and the development

of suitable methods for charging and arbitrating the allocation of resources.

Future evolution: The key to the provision for future evolution is the next header

field. If non-zero, it defines the type of an extension header that is included in the

packet. There are currently extension header types that provide additional data for

special services of the following types: information for routers, route definition,

fragment handling, authentication, encryption and destination handling. Each

extension header type has a specific size and a defined format. Further extension

header types will be defined as new service requirements arise. An extension header,

if present, follows the basic header and precedes the payload and includes a next

header field, enabling multiple extension headers to be employed.

Multicast and anycast: Both IPv4 and IPv6 include support for the transmission of

IP packets to multiple hosts using a single address (one that is in the range reserved

for the purpose). The IP routers are then responsible for routing the packet to all of

the hosts that have subscribed to the group identified by the relevant address. Further

details on IP multicast communication can be found in Section 4.4.1. In addition,

IPv6 supports a new mode of transmission called anycast. This service delivers a

packet to at least one of the hosts that subscribes to the relevant address.

Security: Up to now, Internet applications that require authenticated or private data

transmission have relied on the use of cryptographic techniques in the application

layer. The end-to-end argument supports the view that this is the right place for it. If

security is implemented at the IP level, then users and application developers depend136 CHAPTER 3 NETWORKING AND INTERNETWORKING

upon the correctness of the code that implements it in each router along the way, and

they must trust the routers and other intermediate nodes to handle cryptographic keys.

The advantage of implementing security at the IP level is that it can be applied

without the need for security-aware implementations of application programs. For

example, system managers can implement it in a firewall and apply it uniformly to

all external communication without incurring the cost of encryption for internal

communication. Routers may also exploit an IP-level security mechanism to secure

the routing table update messages that they exchange between themselves.

Security in IPv6 is implemented through the authentication and encrypted

security payload extension header types. These implement features equivalent to the

secure channel concept introduced in Section 2.4.3. The payload is encrypted and/or

digitally signed as required. Similar security features are also available in IPv4 using

IP tunnelling between routers or hosts that implement the IPSec specification (see

RFC 2411 [Thayer 1998]).

Migration from IPv4 • The consequences for the existing Internet infrastructure of a

change in its basic protocol are profound. IP is processed in the TCP/IP protocol stack

at every host and in the software of every router. IP addresses are handled in many

application and utility programs. All of these require upgrading to support the new

version of IP, but the change is made inevitable by the forthcoming exhaustion of the

address space provided by IPv4. The IETF working group responsible for IPv6 has

defined a migration strategy – essentially it involves the implementation of ‘islands’ of

IPv6 routers and hosts communicating with other IPv6 islands via tunnels and gradually

merging into larger islands.

As we have noted, IPv6 routers and hosts should have no difficulty in handling

mixed traffic, since the IPv4 address space is embedded in the IPv6 space. All of the

major operating systems (Windows XP, Mac OS X, Linux and other Unix variants)

already include implementations of UDP and TCP sockets (as described in Chapter 4)

over IPv6, enabling applications to be migrated with a simple upgrade.

The theory of this strategy is technically sound, but implementation progress has

been very slow, perhaps because CIDR and NAT have relieved the pressure to a greater

extent than anticipated. This has begun to change in the mobile phone and portable

device markets, though. All of these devices are likely to be Internet-enabled in the near

future and they cannot easily be hidden behind NAT routers. For example, it is projected

that more than a billion IP devices will be deployed in India and China by 2014. Only

IPv6 can address needs such as that.

3.4.5 MobileIP

Mobile computers such as laptops and tablets are connected to the Internet at different

locations as they migrate. In its owner’s office a laptop may be connected to a local

Ethernet connected to the Internet through a router, it may be connected via a mobile

phone while it is in transit by car or train, then it may be attached to an Ethernet at

another site. The user will wish to access services such as email and the Web at any of

these locations.

Simple access to services does not require a mobile computer to retain a single

address, and it may acquire a new IP address at each site; that is the purpose of theSECTION 3.4 INTERNET PROTOCOLS 137

Dynamic Host Configuration Protocol (DHCP), which enables a newly connected

computer to dynamically acquire an IP address in the address range of the local subnet

and discover the addresses of local resources such as a DNS server from a local DHCP

server. It will also need to discover what local services (such as printing, mail delivery

and so on) are available at each site that it visits. Discovery services are a type of naming

service that assist with this; they are described in Chapter 19 (Section 19.2).

There may be files or other resources on the laptop to which others require access,

or the laptop may be running a distributed application such as a share-monitoring service

that receives notifications of specified events, such as stocks that the user holds passing

a preset threshold. If a mobile computer is to remain accessible to clients and resourcesharing applications when it moves between local networks and wireless networks, it

must retain a single IP number, but IP routing is subnet-based. Subnets are at fixed

locations, and the correct routing of packets to them depends upon their position on the

network.

MobileIP is a solution for the latter problem. The solution is implemented

transparently, so IP communication continues normally when a mobile host computer

moves between subnets at different locations. It is based upon the permanent allocation

of a normal IP address to each mobile host on a subnet in its ‘home’ domain.

When the mobile host is connected at its home base, packets are routed to it in the

normal way. When it is connected to the Internet elsewhere, two agent processes take

responsibility for rerouting. The agents are a home agent (HA) and a foreign agent (FA).

These processes run on convenient fixed computers at the home site and at the current

location of the mobile host.

The HA is responsible for holding up-to-date knowledge of the mobile host’s

current location (the IP address by which it can be reached). It does this with the

assistance of the mobile host itself. When a mobile host leaves its home site, it should

inform the HA, and the HA notes the mobile host’s absence. During the absence it will

behave as a proxy; in order to do so, it tells the local routers to cancel any cached records

relating to the mobile host’s IP address. While it is acting as a proxy, the HA responds

to ARP requests concerning the mobile host’s IP address, giving its own local network

address as the network address of the mobile host.

When the mobile host arrives at a new site, it informs the FA at that site. The FA

allocates a ‘care-of address’ to it – a new, temporary IP address on the local subnet. The

FA then contacts the HA, giving it the mobile host’s home IP address and the care-of

address that has been allocated to it.

Figure 3.20 illustrates the MobileIP routing mechanism. When an IP packet

addressed to the mobile host’s home address is received at the home network, it is routed

to the HA. The HA then encapsulates the IP packet in a MobileIP packet and sends it to

the FA. The FA unpacks the original IP packet and delivers it to the mobile host via the

local network to which it is currently attached. Note that the method by which the HA

and the FA reroute the original packet to its intended recipient is an instance of the

tunnelling technique described in Section 3.3.7.

The HA also sends the care-of address of the mobile host to the original sender. If

the sender is MobileIP-enabled, it will note the new address and use it for subsequent

communication with the mobile host, avoiding the overheads of rerouting via the HA. If

it is not, then it will ignore the change of address and subsequent communication will

continue to be rerouted via the HA.138 CHAPTER 3 NETWORKING AND INTERNETWORKING

The MobileIP solution is effective, but hardly efficient. A solution that treats

mobile hosts as first-class citizens would be preferable, allowing them to wander

without giving prior notice and routing packets to them without any tunnelling or rerouting. We should note that this apparently difficult feat is exactly what is achieved by

the cellular phone network – mobile phones do not change their number as they move

between cells, or even between countries. Instead, they simply notify the local cellular

phone base station of their presence from time to time.

3.4.6 TCP and UDP

TCP and UDP provide the communication capabilities of the Internet in a form that is

useful for application programs. Application developers might wish for other types of

transport service, for example to provide real-time guarantees or security, but such

services would generally require more support in the network layer than IPv4 provides.

TCP and UDP can be viewed as a faithful reflection at the application programming

level of the communication facilities that IPv4 has to offer. IPv6 is another story; it will

certainly continue to support TCP and UDP, but it includes capabilities that cannot be

conveniently accessed through TCP and UDP. It may be useful to introduce additional

types of transport service to exploit them, once the deployment of IPv6 is sufficiently

wide to justify their development.

Chapter 4 describes the characteristics of both TCP and UDP from the point of

view of distributed program developers. Here we shall be quite brief, describing only the

functionality that they add to IP.

Use of ports • The first characteristic to note is that, whereas IP supports

communication between pairs of computers (identified by their IP addresses), TCP and

UDP, as transport protocols, must provide process-to-process communication. This is

accomplished by the use of ports. Port numbers are used for addressing messages to

processes within a particular computer and are valid only within that computer. A port

number is a 16-bit integer. Once an IP packet has been delivered to the destination host,

the TCP- or UDP-layer software dispatches it to a process via a specific port at that host.

UDP features • UDP is almost a transport-level replica of IP. A UDP datagram is

encapsulated inside an IP packet. It has a short header that includes the source and

Sender

Home

Figure 3.20 The MobileIP routing mechanism

Mobile host (MH)

Foreign agent (FA)

Internet

agent

First IP packet

addressed to MH

Address of FA

returned to sender

First IP packet

tunnelled to FA

Subsequent IP packets

tunnelled to FASECTION 3.4 INTERNET PROTOCOLS 139

destination port numbers (the corresponding host addresses are present in the IP header),

a length field and a checksum. UDP offers no guarantee of delivery. We have already

noted that IP packets may be dropped because of congestion or network error. UDP adds

no additional reliability mechanisms except the checksum, which is optional. If the

checksum field is non-zero, the receiving host computes a check value from the packet

contents and compares it with the received checksum; packets for which they do not

match are dropped.

Thus UDP provides a means of transmitting messages of up to 64 kbytes in size

(the maximum packet size permitted by IP) between pairs of processes (or from one

process to several in the case of datagrams addressed to IP multicast addresses), with

minimal additional costs or transmission delays above those due to IP transmission. It

incurs no setup costs and it requires no administrative acknowledgement messages. But

its use is restricted to those applications and services that do not require reliable delivery

of single or multiple messages.

TCP features • TCP provides a much more sophisticated transport service. It provides

reliable delivery of arbitrarily long sequences of bytes via stream-based programming

abstraction. The reliability guarantee entails the delivery to the receiving process of all

of the data presented to the TCP software by the sending process, in the same order. TCP

is connection-oriented. Before any data is transferred, the sending and receiving

processes must cooperate in the establishment of a bidirectional communication

channel. The connection is simply an end-to-end agreement to perform reliable data

transmission; intermediate nodes such as routers have no knowledge of TCP

connections, and the IP packets that transfer the data in a TCP transmission do not

necessarily all follow the same route.

The TCP layer includes additional mechanisms (implemented over IP) to meet the

reliability guarantees. These are:

Sequencing: A TCP sending process divides the stream into a sequence of data

segments and transmits them as IP packets. A sequence number is attached to each

TCP segment. It gives the byte number within the stream for the first byte of the

segment. The receiver uses the sequence numbers to order the received segments

before placing them in the input stream at the receiving process. No segment can be

placed in the input stream until all lower-numbered segments have been received and

placed in the stream, so segments that arrive out of order must be held in a buffer until

their predecessors arrive.

Flow control: The sender takes care not to overwhelm the receiver or the intervening

nodes. This is achieved by a system of segment acknowledgements. Whenever a

receiver successfully receives a segment, it records its sequence number. From time

to time the receiver sends an acknowledgement to the sender, giving the sequence

number of the highest-numbered segment in its input stream together with a window

size. If there is a reverse flow of data, acknowledgements are carried in the normal

data segments; otherwise they travel in acknowledgement segments. The window

size field in the acknowledgement segment specifies the quantity of data that the

sender is permitted to send before the next acknowledgement.

When a TCP connection is used for communication with a remote interactive

program, data may be produced in small quantities but in a very bursty manner. For140 CHAPTER 3 NETWORKING AND INTERNETWORKING

example, keyboard input may result in only a few characters per second, but the

characters should be sent sufficiently quickly for the user to see the results of their

typing. This is dealt with by setting a timeout T on local buffering – typically 0.5

seconds. With this simple scheme, a segment is sent to the receiver whenever data

has been waiting in the output buffer for T seconds, or the contents of the buffer reach

the MTU limit. This buffering scheme cannot add more than T seconds to the

interactive delay. Nagle has described another algorithm that produces less traffic

and is more effective for some interactive applications [Nagle 1984]. Nagle’s

algorithm is used in many TCP implementations. Most TCP implementations are

configurable, allowing applications to change the value of T or to select one of

several buffering algorithms.

Because of the unreliability of wireless networks and the resulting frequent loss

of packets, these flow-control mechanisms are not particularly relevant for wireless

communication. This is one of the reasons for the adoption of a different transport

mechanism in the WAP family of protocols for wide area mobile communication.

But the implementation of TCP for wireless networks is also important, and

modifications to the TCP mechanism have been proposed for this purpose

[Balakrishnan et al. 1995, 1996]. The idea is to implement a TCP support component

at the wireless base station (the gateway between wired and wireless networks). The

support component snoops on TCP segments to and from the wireless network,

retransmitting any outbound segments that are not acknowledged rapidly by the

mobile receiver and requesting retransmissions of inbound segments when gaps in

the sequence numbers are noticed.

Retransmission: The sender records the sequence numbers of the segments that it

sends. When it receives an acknowledgement it notes that the segments were

successfully received, and it may then delete them from its outgoing buffers. If any

segment is not acknowledged within a specified timeout, the sender retransmits it.

Buffering: The incoming buffer at the receiver is used to balance the flow between

the sender and the receiver. If the receiving process issues receive operations more

slowly than the sender issues send operations, the quantity of data in the buffer will

grow. Usually it is extracted from the buffer before it becomes full, but ultimately the

buffer may overflow, and when that happens incoming segments are simply dropped

without recording their arrival. Their arrival is therefore not acknowledged and the

sender is obliged to retransmit them.

Checksum: Each segment carries a checksum covering the header and the data in the

segment. If a received segment does not match its checksum, the segment is dropped.

3.4.7 Domain names

The design and implementation of the Domain Name System (DNS) is described in

detail in Chapter 13; we give a brief overview here to complete our discussion of the

Internet protocols. The Internet supports a scheme for the use of symbolic names for

hosts and networks, such as binkley.cs.mcgill.ca or essex.ac.uk. The named entities are

organized into a naming hierarchy. The named entities are called domains and the

symbolic names are called domain names. Domains are organized in a hierarchy that isSECTION 3.4 INTERNET PROTOCOLS 141

intended to reflect their organizational structure. The naming hierarchy is entirely

independent of the physical layout of the networks that constitute the Internet. Domain

names are convenient for human users, but they must be translated to Internet (IP)

addresses before they can be used as communication identifiers. This is the

responsibility of a specific service, the DNS. Application programs pass requests to the

DNS to convert the domain names that users specify into Internet addresses.

The DNS is implemented as a server process that can be run on host computers

anywhere in the Internet. There are at least two DNS servers in each domain, and often

more. The servers in each domain hold a partial map of the domain name tree below their

domain. They must hold at least the portion consisting of all of the domain and host

names within their domain, but they often contain a larger portion of the tree. DNS

servers handle requests for the translation of domain names outside their portion of the

tree by issuing requests to DNS servers in the relevant domains, proceeding recursively

from right to left, resolving the name in segments. The resulting translation is then

cached at the server handling the original request so that future requests for the

resolution of names referring to the same domain will be resolved without reference to

other servers. The DNS would not be workable without the extensive use of caching,

since the ‘root’ name servers would be consulted in almost every case, creating a service

access bottleneck.

3.4.8 Firewalls

Almost all organizations need Internet connectivity in order to provide services to their

customers and other external users and to enable their internal users to access

information and services. The computers in most organizations are quite diverse,

running a variety of operating systems and application software. The security of their

software is even more varied; some of it may include state-of-the-art security, but much

of it will have little or no capability to ensure that incoming communications can be

trusted and outgoing communications are private when required. In summary, in an

intranet with many computers and a wide range of software it is inevitable that some

parts of the system will have weaknesses that expose it to security attacks. Forms of

attack are detailed further in Chapter 11.

The purpose of a firewall is to monitor and control all communication into and out

of an intranet. A firewall is implemented by a set of processes that act as a gateway to

an intranet (Figure 3.21a), applying a security policy determined by the organization.

The aims of a firewall security policy may include any or all of the following:

Service control: To determine which services on internal hosts are accessible for

external access and to reject all other incoming service requests. Outgoing service

requests and the responses to them may also be controlled. These filtering actions can

be based on the contents of IP packets and the TCP and UDP requests that they

contain. For example, incoming HTTP requests may be rejected unless they are

directed to an official web server host.

Behaviour control: To prevent behaviour that infringes the organization’s policies,

is antisocial or has no discernible legitimate purpose and is hence suspected of

forming part of an attack. Some of these filtering actions may be applicable at the IP

or TCP level, but others may require interpretation of messages at a higher level. For142 CHAPTER 3 NETWORKING AND INTERNETWORKING

example, filtering of email ‘spam’ attacks may require examination of the sender’s

email address in message headers or even the message contents.

User control: The organization may wish to discriminate between its users, allowing

some to access external services but inhibiting others from doing so. An example of

user control that is perhaps more socially acceptable than some is to prevent the

acknowledging of software except to users who are members of the system

administration team, in order to prevent virus infection or to maintain software

standards. This particular example would in fact be difficult to implement without

inhibiting the use of the Web by ordinary users.

Another instance of user control is the management of dialup and other

connections provided for offsite users. If the firewall is also the host for modem

connections, it can authenticate the user at connection time and can require the use of

a secure channel for all communication (to prevent eavesdropping, masquerading and

other attacks on the external connection). That is the purpose of the virtual private

network technology described in the next subsection.

The policy has to be expressed in terms of filtering operations that are performed by

filtering processes operating at several different levels:

Figure 3.21 Firewall configurations

Internet

a) Filtering router Router/ Protected intranet

Internet

b) Filtering router and bastion

filter

Internet

c) Screened subnet for bastion R/filter Bastion R/filter

R/filter Bastion

web/ftp

server

web/ftp

server

web/ftp

serverSECTION 3.4 INTERNET PROTOCOLS 143

IP packet filtering: This is a filter process examining individual IP packets. It may

make decisions based on the destination and source addresses. It may also examine

the service type field of IP packets and interpret the contents of the packets based on

the type. For example, it may filter TCP packets based on the port number to which

they are addressed, and since services are generally located at well-known ports, this

enables packets to be filtered based on the service requested. For example, many sites

prohibit the use of NFS servers by external clients.

For performance reasons, IP filtering is usually performed by a process within

the operating system kernel of a router. If multiple firewalls are used, the first may

mark certain packets for more exhaustive examination by a later firewall, allowing

‘clean’ packets to proceed. It is possible to filter based on sequences of IP packets,

for example, to prevent access to an FTP server before a login has been performed.

TCP gateway: A TCP gateway process checks all TCP connection requests and

segment transmissions. When a TCP gateway process is installed, the setting up of

TCP connections can be controlled and TCP segments can be checked for correctness

(some denial of service attacks use malformed TCP segments to disrupt client

operating systems). When desired, they can be routed through an application-level

gateway for content checking.

Application-level gateway: An application-level gateway process acts as a proxy for

an application process. For example, a policy may be desired that allows certain

internal users to make Telnet connections to certain external hosts. When a user runs

a Telnet program on their local computer, it attempts to establish a TCP connection

with a remote host. The request is intercepted by the TCP gateway. The TCP gateway

starts a Telnet proxy process and the original TCP connection is routed to it. If the

proxy approves the Telnet operation (i.e., if the user is authorized to use the requested

host) it establishes another connection to the requested host and relays all of the TCP

packets in both directions. A similar proxy process would run on behalf of each

Telnet client, and similar proxies might be employed for FTP and other services.

A firewall is usually composed of several processes working at different protocol levels.

It is common for firewall duties to be shared by more than one computer for performance

and fault-tolerance reasons. In all of the configurations described below and illustrated

in Figure 3.21, we show a public web and FTP server without protection. It holds only

published information that requires no protection against public access, and its server

software ensures that only authorized internal users can update it.

IP packet filtering is normally done by a router – a computer with at least two

network addresses on separate IP networks – that runs an RIP process, an IP packetfiltering process and as few other processes as possible. The router/filter must run only

trusted software in a manner that enables its enforcement of filtering policies to be

guaranteed. This involves ensuring that no Trojan horse processes can run on it and that

the filtering and routing software have not been modified or tampered with. Figure

3.21(a) shows a simple firewall configuration that relies only on IP filtering and employs

a single router for that purpose. The network configuration in Figure 3.10 includes two

router/filters acting as firewalls of this type for performance and reliability reasons.

They both obey the same filtering policy and the second does not increase the security

of the system.144 CHAPTER 3 NETWORKING AND INTERNETWORKING

When TCP and application-level gateway processes are required, these usually

run on a separate computer, which is known as a bastion. (The term originates from the

construction of fortified castles; it is a protruding watchtower from which the castle may

be defended or defenders may negotiate with those desiring entry.) A bastion computer

is a host that is located inside the intranet protected by an IP router/filter and runs the

TCP and application-level gateways (Figure 3.21b). Like the router/filter, the bastion

must run only trusted software. In a well-secured intranet, proxies must be used for

access to all outside services. Readers may be familiar with the use of proxies for web

access. These are an instance of the use of firewall proxies; they are often constructed in

a manner that integrates a web cache server (described in Chapter 2). This and other

proxies are likely to require substantial processing and storage resources.

Security can be enhanced by employing two router/filters in series, with the

bastion and any public servers located on a separate subnet linking the router/filters

(Figure 3.21c). This configuration has several security advantages:

• If the bastion policy is strict, the IP addresses of hosts in the intranet need not even

be published to the outside world, and the addresses in the outside world need not

be known to internal computers, since all external communication passes through

proxy processes in the bastion, has access to both.

• If the first router/filter is penetrated or compromised, the second, which is

invisible from outside the intranet and hence less vulnerable, remains to pick up

and reject unacceptable IP packets.

Virtual private networks • Virtual private networks (VPNs) extend the firewall

protection boundary beyond the local intranet by the use of cryptographically protected

secure channels at the IP level. In Section 3.4.4, we outlined the IP security extensions

available in IPv6 and IPv4 with IPSec tunnelling [Thayer 1998]. These are the basis for

the implementation of VPNs. They may be used for individual external users or to

implement secure connections between intranets located at different sites using public

Internet links.

For example, a member of staff may need to connect to the organization’s intranet

via an Internet service provider. Once connected, they should have the same capabilities

as a user inside the firewall. This can be achieved if their local host implements IP

security. The local host holds one or more cryptographic keys that it shares with the

firewall, and these are used to establish a secure channel at connection time. Secure

channel mechanisms are described in detail in Chapter 11.

3.5 Case studies: Ethernet, WiFi and Bluetooth

Up to this point we have discussed the principles involved in the construction of

computer networks and we have described IP, the ‘virtual network layer’ of the Internet.

To complete the chapter, we describe the principles and implementations of three actual

networks.

In the early 1980s, the US Institute of Electrical and Electronic Engineers (IEEE)

established a committee to specify a series of standards for local area networks (the 802

Committee [IEEE 1990]), and its subcommittees have produced a series ofSECTION 3.5 CASE STUDIES: ETHERNET, WIFI AND BLUETOOTH 145

specifications that have become the key standards for LANs. In most cases, the

standards are based on pre-existing industry standards that emerged from research done

in the 1970s. The relevant subcommittees and the standards that they have been

published to date are shown in Figure 3.22.

They differ in performance, efficiency, reliability and cost, but they all provide

relatively high-bandwidth networking capabilities over short and medium distances.

The IEEE 802.3 Ethernet standard has largely won the battle for the wired LAN

marketplace, and we describe it in Section 3.5.1 as our representative wired LAN

technology. Although Ethernet implementations are available for several bandwidths,

the principles of operation are identical in all of them.

The IEEE 802.5 Token Ring standard was a significant competitor for much of

the 1990s, offering advantages over Ethernet in terms of efficiency and its support for

bandwidth guarantees, but it has now disappeared from the marketplace. Readers

interested in a brief description of this interesting LAN technology can find one at

www.cdk5.net/networking. The widespread use of Ethernet switches (as opposed to

hubs) has enabled Ethernets to be configured in a manner that offers bandwidth and

latency guarantees (as discussed further in Section 3.5.1, subsection Ethernet for realtime and quality of service critical applications), and this is one reason for its

displacement of token ring technology.

The IEEE 802.4 Token Bus standard was developed for industrial applications

with real-time requirements and is employed in that domain. The IEEE 802.6

Metropolitan Area standard covers distances up to 50 km and is intended for use in

networks that span towns and cities.

The IEEE 802.11 Wireless LAN standard emerged somewhat later but holds a

major position in the marketplace with products from many vendors under the

commercial name WiFi, and is installed in a large proportion of mobile and handheld

computing devices. The IEEE 802.11 standard is designed to support communication at

speeds up to 54 Mbps over distances of up to 150 m between devices equipped with

simple wireless transmitter/receivers. We describe its principles of operation in Section

3.5.2. Further details on IEEE 802.11 networks can be found in Crow et al. [1997] and

Kurose and Ross [2007].

The IEEE 802.15.1 Wireless Personal Area Network standard (Bluetooth) was

based on a technology first developed in 1999 by the Ericsson company to transport lowFigure 3.22 IEEE 802 network standards

IEEE no. Name Title Reference

802.3 Ethernet CSMA/CD Networks (Ethernet) [IEEE 1985a]

802.4 Token Bus Networks [IEEE 1985b]

802.5 Token Ring Networks [IEEE 1985c]

802.6 Metropolitan Area Networks [IEEE 1994]

802.11 WiFi Wireless Local Area Networks [IEEE 1999]

802.15.1 Bluetooth Wireless Personal Area Networks [IEEE 2002]

802.15.4 ZigBee Wireless Sensor Networks [IEEE 2003]

802.16 WiMAX Wireless Metropolitan Area Networks [IEEE 2004a]146 CHAPTER 3 NETWORKING AND INTERNETWORKING

bandwidth digital voice and data between devices such as tablets, mobile phones and

headsets and was subsequently standardized in 2002 as IEEE 802.15.1. Section 3.5.3

contains a description of Bluetooth.

IEEE 802.15.4 (ZigBee) is another WPAN standard aimed at providing data

communication for very low-bandwidth low-energy devices in the home such as remote

controls, burglar alarm and heating system sensors, and ubiquitous devices such as

active badges and tag readers. Such networks are termed wireless sensor networks and

their applications and communication characteristics are discussed in Chapter 19.

The IEEE 802.16 Wireless MAN standard (commercial name: WiMAX) was

ratified in 2004 and 2005. The IEEE 802.16 standard is designed as an alternative to

cable and DSL links for the ‘last mile’ connection to homes and offices. A variant of the

standard is intended to supersede 802.11 WiFi networks as the main connection

technology for laptop computers and mobile devices in outdoor and indoor public areas.

The ATM technology emerged from major research and standardization efforts in

the telecommunications and computer industries in the late 1980s and early 1990s

[CCITT 1990]. Its purpose is to provide a high-bandwidth wide area digital networking

technology suitable for telephone, data and multimedia (high-quality audio and video)

applications. Although the uptake has been slower than expected, ATM is now the

dominant technology for very high speed wide area networking. It was also seen in some

quarters as a replacement for Ethernet in LAN applications, but it has been less

successful in that marketplace due to competition from 100 Mbps and 1000 Mbps

Ethernets, which are available at much lower cost. Further details on ATM and on other

high-speed network technologies can be found in Tanenbaum [2003] and Stallings

[2002].

3.5.1 Ethernet

The Ethernet was developed at the Xerox Palo Alto Research Center in 1973 [Metcalfe

and Boggs 1976; Shoch et al. 1982, 1985] as part of the programme of research carried

out there on personal workstations and distributed systems. The pilot Ethernet was the

first high-speed local network, demonstrating the feasibility and usefulness of highspeed local networks linking computers on a single site, allowing them to communicate

at high transmission speeds with low error rates and without switching delays. The

original prototype Ethernet ran at 3 Mbps. Ethernet systems are now available with

bandwidths ranging from 10 Mbps to 1000 Mbps.

We shall describe the principles of operation of the 10 Mbps Ethernet specified in

IEEE Standard 802.3 [IEEE 1985a]. This was the first widely deployed local area

network technology. The 100 Mbps variant is now more commonly used; its principles

of operation are identical. We conclude this section with a list of the more important

variants of Ethernet transmission technology and bandwidth that are available. For

comprehensive descriptions of the Ethernet in all its variations, see Spurgeon [2000].

A single Ethernet is a simple or branching bus-like connection line using a

transmission medium consisting of one or more continuous segments of cable linked by

hubs or repeaters. Hubs and repeaters are simple devices that link pieces of wire,

enabling the same signals to pass through all of them. Several Ethernets can be linked at

the Ethernet network protocol level by Ethernet switches or bridges. Switches andSECTION 3.5 CASE STUDIES: ETHERNET, WIFI AND BLUETOOTH 147

bridges operate at the level of Ethernet frames, forwarding them to adjacent Ethernets

when their destination is there. Linked Ethernets appear as a single network to higher

protocol layers, such as IP (see Figure 3.10, where the IP subnets 138.37.88 and

138.37.94 are each composed of several Ethernets linked by components marked

Eswitch). In particular, the ARP protocol (Section 3.4.2) is able to resolve IP addresses

to Ethernet addresses across linked sets of Ethernets; each ARP request is broadcast on

all of the linked networks in a subnet.

The method of operation of Ethernets is defined by the phrase ‘carrier sensing,

multiple access with collision detection’ (abbreviated: CSMA/CD) and they belong to

the class of contention bus networks. Contention buses use a single transmission

medium to link all of the hosts. The protocol that manages access to the medium is called

a medium access control (MAC) protocol. Because a single link connects all hosts, the

MAC protocol combines the functions of a data link layer protocol (responsible for the

transmission of packets on communication links) and a network protocol (responsible

for delivery of packets to hosts) in a single protocol layer.

Packet broadcasting • The method of communication in CSMA/CD networks is by

broadcasting packets of data on the transmission medium. All stations are continuously

‘listening' to the medium for packets that are addressed to them. Any station wishing to

transmit a message broadcasts one or more packets (called frames in the Ethernet

specification) on the medium. Each packet contains the address of the destination

station, the address of the sending station and a variable-length sequence of bits

representing the message to be transmitted. Data transmission proceeds at 10 Mbps (or

at the higher speeds specified for 100 and 1000 Mbps Ethernets) and packets vary in

length between 64 and 1518 bytes, so the time required to transmit a packet on a 10

Mbps Ethernet is 50–1200 microseconds, depending on its length. The MTU is specified

as 1518 bytes in the IEEE standard, although there is no technical reason for any

particular fixed limit except the need to limit delays caused by contention.

The address of the destination station normally refers to a single network

interface. Controller hardware at each station receives a copy of every packet. It

compares the destination address in each packet with a wired-in local address, ignoring

packets addressed to other stations and passing those with a matching address to the

local host. The destination address may also specify a broadcast or a multicast address.

Ordinary addresses are distinguished from broadcast and multicast addresses by their

higher-order bit (0 and 1, respectively). An address consisting of all 1s is reserved for

use as a broadcast address and is used when a message is to be received by all of the

stations on the network. This is used, for example, to implement the ARP IP address

resolution protocol. Any station that receives a packet with a broadcast address will pass

it on to its local host. A multicast address specifies a limited form of broadcast that is

received by a group of stations whose network interfaces have been configured to

receive packets with that multicast address. Not all implementations of Ethernet network

interfaces can recognize multicast addresses.

The Ethernet network protocol (providing for the transmission of Ethernet packets

between pairs of hosts) is implemented in the Ethernet hardware interface; protocol

software is required for the transport layer and those above it.148 CHAPTER 3 NETWORKING AND INTERNETWORKING

Ethernet packet layout • The packets (or more correctly, frames) transmitted by stations

on the Ethernet have the following layout:

Apart from the destination and source addresses already mentioned, frames include a

fixed 8-byte prefix, a length field, a data field and a checksum. The prefix is used for

hardware timing purposes and consists of a preamble of 7 bytes, each containing the bit

pattern 10101010 followed by a single-byte start frame delimiter (S in the diagram) with

the pattern 10101011.

Despite the fact that the specification does not allow more than 1024 stations on a

single Ethernet, addresses occupy 6 bytes, providing 248 different addresses. This

enables every Ethernet hardware interface to be given a unique address by its

manufacturer, ensuring that all of the stations in any interconnected set of Ethernets will

have unique addresses. The US Institute of Electrical and Electronic Engineers (IEEE)

acts as an allocation authority for Ethernet addresses, allocating separate ranges of 48-

bit addresses to the manufacturers of Ethernet hardware interfaces. These are referred to

as MAC addresses, since they are used by the medium access control layer. In fact,

MAC addresses allocated in this fashion have also been adopted as unique addresses for

use in other network types in the IEEE 802 family, including 802.11 (WiFi) and

802.15.1 (Bluetooth).

The data field contains all or part (if the message length exceeds 1500 bytes) of

the message that is being transmitted. The lower bound of 46 bytes on the data field

ensures a minimum packet length of 64 bytes, which is necessary in order to guarantee

that collisions will be detected by all stations on the network, as explained below.

The frame check sequence is a checksum generated and inserted by the sender and

used to validate packets by the receiver. Packets with incorrect checksums are simply

dropped by the data link layer in the receiving station. This is another example of the

application of the end-to-end argument: to guarantee the transmission of a message, a

transport-layer protocol such as TCP, which acknowledges receipt of each packet and

retransmits any unacknowledged packets, must be used. The incidence of data

corruption in local networks is so small that the use of this method of recovery when

guaranteed delivery is required is entirely satisfactory and it enables a less costly

transport protocol such as UDP to be employed when there is no need for delivery

guarantees.

Packet collisions • Even in the relatively short time that it takes to transmit packets

there is a finite probability that two stations on the network will attempt to transmit

messages simultaneously. If a station attempts to transmit a packet without checking

whether the medium is in use by other stations, a collision may occur.

The Ethernet has three mechanisms to deal with this possibility. The first is called

carrier sensing: the interface hardware in each station listens for the presence of a signal

(known as the carrier by analogy with radio broadcasting) in the medium. When a

station wishes to transmit a packet, it waits until no signal is present in the medium and

then begins to transmit.

bytes: 7 1 6 6 2 46 < length < 1500 4

Preamble S Destination

address

Source

address

Length

of data

Data for transmission ChecksumSECTION 3.5 CASE STUDIES: ETHERNET, WIFI AND BLUETOOTH 149

Unfortunately, carrier sensing does not prevent all collisions. The possibility of

collision remains due to the finite time τ for a signal inserted at a point in the medium

(travelling at electronic speed: approximately 2 × 108 metres per second) to reach all

other points. Consider two stations A and B that are ready to transmit packets at almost

the same time. If A begins to transmit first, B can check and find no signal in the medium

at any time t < τ after A has begun to transmit. B then begins to transmit, interfering with

A’s transmission. Both A’s packet and B’s packet will be damaged by the interference.

The technique used to recover from such interference is called collision detection.

Whenever a station is transmitting a packet through its hardware output port, it also

listens on its input port and the two signals are compared. If they differ, then a collision

has occurred. When this happens the station stops transmitting and produces a jamming

signal to ensure that all stations recognize the collision. As we have already noted, a

minimum packet length is necessary to ensure that collisions are always detected. If two

stations transmit approximately simultaneously from opposite ends of the network, they

will not become aware of the collision for 2τ seconds (because the first sender must be

still transmitting when it receives the second signal). If the packets that they transmit

take less than τ to be broadcast, the collision will not be noticed, since each sending

station would not see the other packet until after it has finished transmitting its own,

whereas stations at intermediate points would receive both packets simultaneously,

resulting in data corruption.

After the jamming signal, all transmitting and listening stations cancel the current

packet. The transmitting stations then have to try to transmit their packets again. A

further difficulty now arises. If the stations involved in the collision all attempt to

retransmit their packets immediately after the jamming signal, another collision will

probably occur. To avoid this, a technique known as back-off is used. Each of the

stations involved in a collision chooses to wait a time nτ before retransmitting. The value

of n is a random integer chosen separately at each station and bounded by a constant L

defined in the network software. If a further collision occurs, the value of L is doubled

and the process is repeated if necessary for up to 10 attempts.

Finally, the interface hardware at the receiving station computes the check

sequence and compares it with the checksum transmitted in the packet. Using all of these

techniques, the stations connected to the Ethernet are able to manage the use of the

medium without any centralized control or synchronization.

Ethernet efficiency • The efficiency of an Ethernet is the ratio of the number of packets

transmitted successfully as a proportion of the theoretical maximum number that could

be transmitted without collisions. It is affected by the value of τ, since the interval of 2τ

seconds after a packet transmission starts is the ‘window of opportunity’ for collisions

– no collision can occur later than 2τ seconds after a packet starts to be transmitted. It is

also affected by the number of stations on the network and their level of activity.

For a 1 km cable, the value of τ is less than 5 microseconds and the probability of

collisions is small enough to ensure high efficiency. The Ethernet can achieve a channel

utilization of between 80 and 95%, although the delays due to contention become

noticeable when 50% utilization is exceeded. Because the loading is variable, it is

impossible to guarantee the delivery of a given message within any fixed time, since the

network might be fully loaded when the message is ready for transmission. But the150 CHAPTER 3 NETWORKING AND INTERNETWORKING

probability of transferring the message with a given delay is as good as, or better than,

that of other network technologies.

Empirical measurements of the performance of an Ethernet at Xerox PARC,

reported by Shoch and Hupp [1980], confirm this analysis. In practice, the load on

Ethernets used in distributed systems varies quite widely. Many networks are used

primarily for asynchronous client-server interactions, and these operate for most of the

time with no stations waiting to transmit. Their low level of contention results in a

channel utilization close to 1. Networks that support bulk data access for large numbers

of users experience more load, and those that carry multimedia streams are liable to be

overwhelmed if more than a few streams are transmitted concurrently.

Physical implementations • The description above defines the MAC-layer protocol for

all Ethernets. Widespread adoption across a large marketplace has resulted in the

availability of very low-cost controller hardware to perform the algorithms required for

its implementation, and this is included as a standard part of many desktop and consumer

computers.

A wide range of physical Ethernet implementations have been based on it to offer

a variety of performance and cost trade-offs and to exploit increased hardware

performance. The variations result from the use of different transmission media –

coaxial cable, twisted copper wire (similar to telephone wiring) and optical fibre – with

differing limits on transmission range, and from the use of higher signalling speeds,

resulting in greater system bandwidth and generally shorter transmission ranges. The

IEEE has adopted a number of standards for physical-layer implementations, and a

naming scheme is used to distinguish them. Names such as 10Base5 and 100BaseT are

used. They have the following form:

We tabulate the bandwidth and maximum range of various currently available standard

configurations and cable types in Figure 3.23. Configurations ending with the T

designation are implemented with UTP cabling – unshielded twisted wires (telephone

wiring) – and this is organized as a hierarchy of hubs with computers as the leaves of the

tree. In that case, the segment lengths given in our table are twice the maximum

permissible distance from a computer to a hub.

<R><B><L> Where: R = data rate in Mbps

B = medium signalling type (baseband or broadband)

L = maximum segment length in metres/100 or T

(twisted pair cable hierarchy)

Figure 3.23 Ethernet ranges and speeds

10Base5 10BaseT 100BaseT 1000BaseT

Data rate 10 Mbps 10 Mbps 100 Mbps 1000 Mbps

Max. segment lengths:

Twisted wire (UTP) 100 m 100 m 100 m 25 m

Coaxial cable (STP) 500 m 500 m 500 m 25 m

Multi-mode fibre 2000 m 2000 m 500 m 500 m

Mono-mode fibre 25000 m 25000 m 20000 m 2000 mSECTION 3.5 CASE STUDIES: ETHERNET, WIFI AND BLUETOOTH 151

Ethernet for real-time and quality of service critical applications • It is often argued that

the Ethernet MAC protocol is inherently unsuitable for real-time or quality of service

critical applications because of its lack of a guaranteed delivery delay. But it should be

noted that most Ethernet installations are now based on the use of MAC-level switches,

as illustrated in Figure 3.10 and described in Section 3.3.7 (rather than hubs or cables

with a tap for each connection, as was formerly the case). The use of switches

throughout results in a separate segment for each host with no packets transmitted on it

other than those addressed to that host. Hence if traffic to the host is from a single source,

there is no contention for the medium – efficiency is 100% and latency is constant. The

possibility of contention arises only at the switches, and these can be, and often are,

designed to handle several packets concurrently. Hence a lightly loaded switched

Ethernet installation approximates to 100% efficiency with a constant low latency, and

they are therefore often successfully used in these critical application areas.

A further step towards real-time support for Ethernet-style MAC protocols is

described in [Rether; Pradhan and Chiueh 1998] and a similar scheme is implemented

in an open-source Linux extension [RTnet]. These software approaches address the

contention problem by implementing an application-level cooperative protocol to

reserve timeslots for the use of the medium. This protocol depends upon the cooperation

of all the hosts connected to a segment.

3.5.2 IEEE 802.11 (WiFi) wireless LAN

In this section, we summarize the special characteristics of wireless networking that

must be addressed by a wireless LAN technology and explain how IEEE 802.11

addresses them. The IEEE 802.11 standard extends the carrier-sensing multiple access

(CSMA) principle employed by Ethernet (IEEE 802.3) technology to suit the

characteristics of wireless communication. The 802.11 standard is intended to support

communication between computers located within about 150 metres of each other at

speeds up to 54 Mbps.

Figure 3.24 illustrates a portion of an intranet including a wireless LAN. Several

mobile wireless devices communicate with the rest of the intranet through a base station

that is an access point to the wired LAN. A wireless network that connects to the world

through an access point to a conventional LAN is known as an infrastructure network.

An alternative configuration for wireless networking is known as an ad hoc

network. Ad hoc networks do not include an access point or base station. They are built

‘on the fly’ as a result of the mutual detection of two or more mobile devices with

wireless interfaces in the same vicinity. An ad hoc network might occur, for example,

when two or more laptop users in a room initiate a connection to any available station.

They might then share files by launching a file server process on one of the machines.

At the physical level, IEEE 802.11 networks use radio frequency signals (in the

licence-free 2.4 GHz and 5 GHz bands) or infrared signalling as the transmission

medium. The radio version of the standard has received the most commercial attention,

and we shall describe that. The IEEE 802.11b standard was the first variant to see

widespread use. It operates in the 2.4 GHz band and supports data communication at up

to 11 Mbps. It has been installed from 1999 onwards with base stations in many offices,

homes and public places, enabling laptop computers and handheld devices to access

local networked devices or the Internet. IEEE 802.11g is a more recent enhancement of152 CHAPTER 3 NETWORKING AND INTERNETWORKING

802.11b that uses the same 2.4 GHz band but a different signalling technique to achieve

speeds up to 54 Mbps. Finally, the 802.11a variant works in the 5 GHz band and delivers

a more certain 54 Mbps of bandwidth over a somewhat shorter range. All variants use

various frequency-selection and frequency-hopping techniques to avoid external

interference and mutual interference between independent wireless LANs, which we

shall not detail here. We focus instead on the changes to the CSMA/CD mechanism that

are needed in the MAC layer for all versions of 802.11 to enable broadcast data

transmission to be used with radio transmission.

Like Ethernet, the 802.11 MAC protocol offers equal opportunities to all stations

to use the transmission channel, and any station may transmit directly to any other. A

MAC protocol controls the use of the channel by the various stations. As for the

Ethernet, the MAC layer also performs the functions of both a data link layer and a

network layer, delivering data packets to the hosts on a network.

Several problems arise from the use of radio waves rather than wires as the

transmission medium. These problems stem from the fact that the carrier-sensing and

collision-detection mechanisms employed in Ethernets are effective only when the

strength of signals is approximately the same throughout a network.

We recall that the purpose of carrier sensing is to determine whether the medium

is free at all points between the sending and receiving stations, and that of collision

detection is to determine whether the medium in the vicinity of the receiver is free from

interference during the transmission. Because signal strength is not uniform throughout

the space in which wireless LANs operate, carrier detection and collision detection may

fail in the following ways:

Hidden stations: Carrier sensing may fail to detect that another station on the

network is transmitting. This is illustrated in Figure 3.24. If tablet D is transmitting

to the base station E, laptop A may not be able to sense D’s signal because of the radio

obstruction shown. A might then start transmitting, causing a collision at E unless

steps are taken to prevent this.

Figure 3.24 Wireless LAN configuration

LAN

Server

Wireless

LAN

Laptops

Base station/

access point

Tablet

Radio obstruction

A B C

D

ESECTION 3.5 CASE STUDIES: ETHERNET, WIFI AND BLUETOOTH 153

Fading: Due to the inverse square law of electromagnetic wave propagation, the

strength of radio signals diminishes rapidly with the distance from the transmitter.

Stations within a wireless LAN may be out of range of other stations in the same

LAN. Thus in Figure 3.24, laptop A may not be able to detect a transmission by C,

although each of them can transmit successfully to B or E. Fading defeats both carrier

sensing and collision detection.

Collision masking: Unfortunately, the ‘listening’ technique used in the Ethernet to

detect collisions is not very effective in radio networks. Because of the inverse square

law referred to above, the locally generated signal will always be much stronger than

any signal originating elsewhere, effectively drowning out the remote transmission.

So, laptops A and C might both transmit simultaneously to E and neither would detect

that collision, but E would receive only a garbled transmission.

Despite its fallibility, carrier sensing is not dispensed with in IEEE 802.11 networks;

rather, it is augmented by the addition of a slot reservation mechanism to the MAC

protocol. The resulting scheme is called carrier sensing, multiple access with collision

avoidance (CSMA/CA).

When a station is ready to transmit, it senses the medium. If it detects no carrier

signal it may assume that one of the following conditions is true:

1. The medium is available.

2. An out-of-range station is in the process of requesting a slot.

3. An out-of-range station is using a slot that it had previously reserved.

The slot-reservation protocol involves the exchange of a pair of short messages (frames)

between the intending sender and the receiver. The first is a request to send (RTS) frame

from the sender to the receiver. The RTS message specifies a duration for the slot

requested. The receiver replies with a clear to send (CTS) frame, repeating the duration

of the slot. The effect of this exchange is as follows:

• Stations within range of the sender will pick up the RTS frame and take note of

the duration.

• Stations within range of the receiver will pick up the CTS frame and take note of

the duration.

As a result, all of the stations within range of both the sender and the receiver will refrain

from transmitting for the duration of the requested slot, leaving the channel free for the

sender to transmit a data frame of the appropriate length. Finally, successful receipt of

the data frame is acknowledged by the receiver to help deal with the problem of external

interference with the channel. The slot-reservation feature of the MAC protocol helps to

avoid collisions in these ways:

• The CTS frames help to avoid the hidden station and fading problems.

• The RTS and CTS frames are short, so the risk of collisions with them are low. If

one is detected, or an RTS does not result in a CTS, a random back-off period is

used, as in Ethernet.154 CHAPTER 3 NETWORKING AND INTERNETWORKING

• When the RTS and CTS frames have been correctly exchanged, there should be

no collisions involving the subsequent data and acknowledgement frames unless

intermittent fading prevented a third party from receiving either of them.

Security • The privacy and integrity of communication is an obvious concern for

wireless networks. Any station that is within range and equipped with a

receiver/transmitter might seek to join a network, or, failing that, it might eavesdrop on

transmissions between other stations. The first attempt to address the security issues for

802.11 is entitled Wired Equivalent Privacy (WEP). Unfortunately, WEP is anything but

what its name implies. Its security design was flawed in several ways that enabled it to

be broken fairly easily. We describe its weaknesses and summarize the Wi-Fi Protected

Access (WPA) system that succeeded it in Section 11.6.4.

3.5.3 IEEE 802.15.1 Bluetooth wireless PAN

Bluetooth is a wireless personal area network technology that emerged from the need to

link mobile phones, laptop computers and other personal devices without wires. A

special interest group (SIG) of mobile phone and computer manufacturers led by L.M.

Ericsson developed a specification for a wireless personal area network (WPAN) for the

transmission of digital voice streams as well as data [Haartsen et al. 1998]. Version 1.0

of the Bluetooth standard was published in 1999, borrowing its name from a Viking

king. We describe Version 1.1 here. It was published in 2002 resolving some problems.

The IEEE 802.15 Working Group then adopted it as standard 802.15.1 and published a

specification for the physical and data link layers [IEEE 2002].

Bluetooth networks differ substantially from IEEE 802.11 (WiFi), the only other

widely adopted wireless networking standard, in ways that reflect the different

application requirements of WPANs and the different cost and energy consumption

targets for which they are designed. Bluetooth aims to support very small, low-cost

devices such as ear-mounted wireless headsets receiving digital audio streams from a

mobile phone as well as interconnections between computers, phones, tablets and other

mobile devices. The cost target was to add only five dollars to the cost of a handheld

device and the energy target to utilize only a small fraction of the total battery power

used by a phone or tablet, enabling operation for several hours even with lightweight

batteries used in wearable devices such as headsets.

The intended applications require less bandwidth and a shorter transmission range

than typical wireless LAN applications. This is fortunate because Bluetooth operates in

the same crowded 2.4 GHz licence-free frequency band as WiFi networks, cordless

phones and many emergency service communication systems. Transmission is at low

energy, hopping at a rate of 1600 times per second between 79 1 MHz sub-bands of the

permitted frequency band to minimize the effects of interference. The output power of

normal Bluetooth devices is 1 milliwatt, giving a coverage of only 10 metres; 100

milliwatt devices with a range of up to 100 metres are permitted for applications such as

home networks. Energy efficiency is further improved by the inclusion of an adaptive

range facility, which adjusts the transmitted power to a lower level when partner devices

are nearby (as determined by the strength of the signals initially received).

Bluetooth nodes associate dynamically in pairs with no prior knowledge required.

The protocol for association is described below. After a successful association theEXERCISES 155

initiating node has the role of master and the other slave. A Piconet is a dynamically

associated network composed of one master and up to seven active slaves. The master

controls the use of the communication channel, allocating timeslots to each slave. A

node that is in more than one Piconet can act as a bridge, enabling the masters to

communicate – multiple Piconets linked in this fashion are termed a scatternet. Most

types of device have the capacity to act as either master or slave.

All Bluetooth nodes are also equipped with a globally unique 48-bit MAC address

(see Section 3.5.1), although it is only the master’s MAC address that is used in the

protocol. When a slave becomes active in a Piconet, it is assigned a temporary local

address in the range 1 to 7 to reduce the length of packet headers. In addition to the seven

active slaves, a Piconet may contain up to 255 parked nodes in low-power mode

awaiting an activation signal from the master.

Association protocol • To conserve energy, devices remain in sleep or standby mode

before any associations are made or when no recent communication has occurred. In

standby mode they wake to listen for activation messages at intervals ranging from 0.64

to 2.56 seconds. To associate with a known nearby node (parked), the initiating node

transmits a train of 16 page packets, on 16 frequency sub-bands, which may have to be

repeated several times. To contact any unknown node within range, the initiator must

first broadcast a train of inquiry messages. These transmission trains can occupy up to

about 5 seconds in the worst case, leading to a maximum association time of 7–10

seconds.

Association is followed by an optional authentication exchange based on usersupplied or previously received authentication tokens, to ensure that the association is

with the intended node and not an imposter. A slave then remains synchronized to the

master by observing regularly transmitted packets from the master, even when they are

not addressed to the slave. A slave that is inactive can be placed in parked mode by the

master, freeing its slot in the Piconet for use by another node.

The requirement to support synchronous communication channels with adequate

quality of service for the transmission of two-way real-time audio (for example, between

a phone and its owner’s wireless headset) as well as asynchronous communication for

data exchange dictated a network architecture very different from the best-effort

multiple-access design of Ethernet and WiFi networks. Synchronous communication is

achieved by the use of a simple two-way communication protocol between a master and

one of its slaves, termed a synchronous connection oriented (SCO) link on which master

and slave must send alternating synchronized packets. Asynchronous communication is

achieved by an asynchronous connection-less (ACL) link on which the master sends

asynchronous poll packets to its slaves periodically and the slaves transmit only after

receiving a poll.

All variants of the Bluetooth protocol use frames that fit within the structure

shown in Figure 3.25. Once a Piconet has been established, the access code consists of

a fixed preamble to synchronize the sender and receiver and identify the start of a slot,

followed by a code derived from the master’s MAC address that uniquely identifies the

Piconet. The latter ensures that frames are correctly routed in situations where there are

multiple overlapping Piconets. Because the medium is likely to be noisy and real-time

communication cannot rely on retransmission, each bit in the header is transmitted in

triplicate to provide redundancy for both the information and the checksum parts.156 CHAPTER 3 NETWORKING AND INTERNETWORKING

The address field is just 3 bits to allow addressing to any of the seven currently

active slaves. A zero address from the master indicates a broadcast. There are single-bit

fields for flow control, acknowledgement and sequence numbering. The flow-control bit

is used by a slave to indicate to the master that its buffers are full; the master should

await a frame with a non-zero acknowledgement bit from the slave. The sequence

number bit is inverted on each new frame sent to the same node; this enables duplicate

(i.e., retransmitted) frames to be detected.

SCO links are used in time-critical applications such as the transmission of a twoway voice conversation. Packets must be short to keep the latency low, and there is little

purpose in reporting or retransmitting corrupted packets in such applications since the

retransmitted data would arrive too late to be useful. So the SCO protocol uses a simple,

highly redundant protocol in which 80 bits of voice data are normally transmitted in

triplicate to produce a 240-bit payload. Any two matching 80-bit replicas are taken as

valid.

On the other hand, ACL links are used for data-transfer applications such as

address book synchronization between a computer and a phone with a larger payload.

The payload is not replicated but may contain an internal checksum that is checked at

the application level, and in the case of failure retransmission can be requested.

Data is transmitted in packets occupying timeslots of 625 microseconds clocked

and allocated by the master node. Each packet is transmitted on a different frequency in

a hopping sequence defined by the master node. Because these slots are not large enough

to allow a substantial payload, frames may be extended to occupy one, three or five slots.

These characteristics and the underlying physical transmission method result in a

maximum total throughput of 1 Mbps for a Piconet, accommodating up to three

synchronous duplex channels of 64 Kpbs between a master and its slaves or a channel

for asynchronous data transfer at rates up to 723 Kbps. These throughputs are calculated

for the most redundant version of the SCO protocol, as described above. Other protocol

variants are defined that trade the robustness and simplicity (and therefore low

computational cost) of triplicated data for higher throughput.

Unlike most network standards, Bluetooth includes specifications (called profiles)

for several application-level protocols, some of which are very specific to particular

applications. The purpose of these profiles is to increase the likelihood that devices

Figure 3.25 Bluetooth frame structure

Header

SCO packets (e.g., for voice data) have a 240-bit payload containing 80 bits of data triplicated, filling

exactly one timeslot.

bits: 72 54 0 – 2744

Access code Header (redundantly encoded) Data for transmission

bits: 3 1 1 1 4 8

Destination Flow Ack Seq Type Header checksum

Address within

Piconet

= ACL, SCO,

poll, nullSECTION 3.6 SUMMARY 157

manufactured by different vendors will interwork. Thirteen application profiles are

covered: generic access, service discovery, serial port, generic object exchange, LAN

access, dialup networking, fax, cordless telephony, intercom, headset, object push, file

transfer and synchronization. Others are in preparation, including ambitious attempts to

transmit high-quality music and even video over Bluetooth.

Bluetooth occupies a special niche in the range of wireless local networks. It

achieves its ambitious design goal of supporting synchronous real-time audio

communication with satisfactory quality of service (see Chapter 20 for further

discussion of quality of service issues) as well as asynchronous data transfer using very

low cost, compact and portable hardware, low power and very limited bandwidth.

Its principal limitation is the time taken (up to 10 seconds) for association of new

devices. This impedes its use for certain applications, especially where devices are

moving relative to each other, preventing its use, for example, to pay road tolls or to

transmit promotional information to mobile phone users as they pass a store. A useful

further reference on Bluetooth networking is the book by Bray and Sturman [2002].

Version 2.0 of the Bluetooth standard, with data throughputs up to 3 Mbps –

sufficient to carry CD-quality audio – was released in 2004. Other improvements

included a faster association mechanism and larger Piconet addresses. Versions 3 and 4

of the standard were under development at the time of writing. Version 3 integrates a

Bluetooth control protocol with a WiFi data transfer layer to achieve throughputs up to

24 Mbps. Version 4 is under development as an ultra-low power Bluetooth technology

for devices requiring a very long battery life.

3.6 Summary

We have focused here on the networking concepts and techniques that are needed as a

basis for distributed systems, approaching them from the point of view of a distributed

system designer. Packet networks and layered protocols provide the basis for

communication in distributed systems. Local area networks are based on packet

broadcasting on a shared medium; Ethernet is the dominant technology. Wide area

networks are based on packet switching to route packets to their destinations through a

connected network. Routing is a key mechanism and a variety of routing algorithms are

used, of which the distance-vector method is the most basic but effective. Congestion

control is needed to prevent overflow of buffers at the receiver and at intermediate

nodes.

Internetworks are constructed by layering a ‘virtual’ internetwork protocol over

collections of networks linked together by routers. The Internet TCP/IP protocols enable

computers in the Internet to communicate with one another in a uniform manner,

irrespective of whether they are on the same local area network or in different countries.

The Internet standards include many application-level protocols that are suitable for use

in wide area distributed applications. IPv6 has the much larger address space needed for

the future evolution of the Internet and provision for new application requirements such

as quality of service and security.

Mobile users are supported by MobileIP for wide area roaming and by wireless

LANs based on IEEE 802 standards for local connectivity.158 CHAPTER 3 NETWORKING AND INTERNETWORKING

EXERCISES

3.1 A client sends a 200 byte request message to a service, which produces a response

containing 5000 bytes. Estimate the total time required to complete the request in each

of the following cases, with the performance assumptions listed below:

i) using connectionless (datagram) communication (for example, UDP);

ii) using connection-oriented communication (for example, TCP);

iii) when the server process is in the same machine as the client.

[Latency per packet (local or remote,

incurred on both send and receive): 5 ms

Connection setup time (TCP only): 5 ms

Data transfer rate: 10 Mbps

MTU: 1000 bytes

Server request processing time: 2 ms

Assume that the network is lightly loaded.]

pages 98, 138

3.2 Describe the network type that is used in place of wired LANs to provide connectivity

for mobile devices. pages 104

3.3 What is the use of a switching system? What are the different types of switching used in

computer networks? pages 107

3.4 Make a table similar to Figure 3.5 describing the work done by the software in each

protocol layer when Internet applications and the TCP/IP suite are implemented over an

Ethernet. pages 110, 138, 146

3.5 What does the term “protocol” mean? What is the role of protocol layers in a network

system? Describe the role of the protocols used in the presentation layer, with

examples. pages 108, 109, 111

3.6 How does adaptive routing ensure the best route of communication between two points

in the network? page 114

3.7 Compare connectionless (UDP) and connection-oriented (TCP) communication for the

implementation of each of the following application-level or presentation-level

protocols:

i) virtual terminal access (for example, Telnet);

ii) file transfer (for example, FTP);

iii) user location (for example, rwho, finger);

iv) information browsing (for example, HTTP);

v) remote procedure call.

page 138

3.8 Explain how TCP ensures the reliable delivery of long sequences of bytes via streambased programming abstraction. Why can’t this happen in a UDP? pages 138, 139EXERCISES 159

3.9 A specific problem that must be solved in remote terminal access protocols such as

Telnet is the need to transmit exceptional events such as ‘kill signals’ from the ‘terminal’

to the host in advance of previousl transmitted data. Kill signals should reach their

destination ahead of any other ongoing transmissions. Discuss the solution of this

problem with connection-oriented and connectionless protocols. page 138

3.10 Explain the following issues for a network with Ethernet transmission technology:

i) packet layout;

ii) packet collision resolution;

iii) efficiency.

pages 147, 148, 149

3.11 For an organization that has Internet connectivity, suggest a scheme that would improve

its security and enable it to provide services to its customers and other external users,

while also allowing internal users to access information and services.

page 141

3.12 Show the sequence of changes to the routing tables in Figure 3.8 that will occur

(according to the RIP algorithm given in Figure 3.9) after the link labelled 3 in Figure

3.7 is broken. pages 114–117

3.13 Use the diagram in Figure 3.13 as a basis for an illustration showing the segmentation

and encapsulation of an HTTP request to a server and the resulting reply. Assume that

the request is a short HTTP message, but the reply includes at least 2000 bytes of

HTML. page 109, 123

3.14 Consider the use of TCP in a Telnet remote terminal client. How should the keyboard

input be buffered at the client? Investigate Nagle’s and Clark’s algorithms [Nagle 1984,

Clark 1982] for flow control and compare them with the simple algorithm described on

page 103 when TCP is used by:

a) a web server,

b) a Telnet application,

c) a remote graphical application with continuous mouse input.

pages 118, 139

3.15 Construct a network diagram similar to Figure 3.10 for the local network at a shopping

mall or for your college campus. page 120

3.16 Describe how you would configure a firewall to protect the local network at your

institution or company. What incoming and outgoing requests should it intercept?

page 141

3.17 Consider a system where the underlying network is an Ethernet. How does the

address resolution module convert the 32-bit Internet address to a 48-bit Ethernet

address? page 128

3.18 Do all the computers and devices that access the Internet in a LAN need to be

assigned globally unique IP addresses? What mechanism is available to deal with

such issues? page 132This page intentionally left blank161

4

INTERPROCESS COMMUNICATION

4.1 Introduction

4.2 The API for the Internet protocols

4.3 External data representation and marshalling

4.4 Multicast communication

4.5 Network virtualization: Overlay networks

4.6 Case study: MPI

4.7 Summary

This chapter is concerned with the characteristics of protocols for communication

between processes in a distributed system – that is, interprocess communication.

Interprocess communication in the Internet provides both datagram and stream

communication. The Java APIs for these are presented, together with a discussion of their

failure models. They provide alternative building blocks for communication protocols.

This is complemented by a study of protocols for the representation of collections of data

objects in messages and of references to remote objects. Together, these services offer

support for the construction of higher-level communication services, as discussed in the

following two chapters.

The interprocess communication primitives discussed above all support point-topoint communication, yet it is equally useful to be able to send a message from one sender

to a group of receivers. The chapter also considers multicast communication, including

IP multicast and the key concepts of reliability and ordering of messages in multicast

communication.

Multicast is an important requirement for distributed applications and must be

provided even if underlying support for IP multicast is not available. This is typically

provided by an overlay network constructed on top of the underlying TCP/IP network.

Overlay networks can also provide support for file sharing, enhanced reliability and

content distribution.

The Message Passing Interface (MPI) is a standard developed to provide an API for

a set of message-passing operations with synchronous and asynchronous variants.162 CHAPTER 4 INTERPROCESS COMMUNICATION

4.1 Introduction

This and the next two chapters are concerned with the communication aspects of

middleware, although the principles discussed are more widely applicable. This one is

concerned with the design of the components shown in the darker layer in Figure 4.1.

The layer above it is discussed in Chapter 5, which examines remote invocation, and

Chapter 6, which is concerned with indirect communications paradigms.

Chapter 3 discussed the Internet transport-level protocols UDP and TCP without

saying how middleware and application programs could use these protocols. The next

section of this chapter introduces the characteristics of interprocess communication and

then discusses UDP and TCP from a programmer’s point of view, presenting the Java

interface to each of these two protocols, together with a discussion of their failure

models.

The application program interface to UDP provides a message passing abstraction

– the simplest form of interprocess communication. This enables a sending process to

transmit a single message to a receiving process. The independent packets containing

these messages are called datagrams. In the Java and UNIX APIs, the sender specifies

the destination using a socket – an indirect reference to a particular port used by the

destination process at a destination computer.

The application program interface to TCP provides the abstraction of a two-way

stream between pairs of processes. The information communicated consists of a stream

of data items with no message boundaries. Streams provide a building block for

producer-consumer communication. A producer and a consumer form a pair of

processes in which the role of the first is to produce data items and the role of the second

is to consume them. The data items sent by the producer to the consumer are queued on

arrival at the receiving host until the consumer is ready to receive them. The consumer

must wait when no data items are available. The producer must wait if the storage used

to hold the queued data items is exhausted.

Section 4.3 is concerned with how the objects and data structures used in

application programs can be translated into a form suitable for sending messages over

the network, taking into account the fact that different computers may use different

representations for simple data items. It also discusses a suitable representation for

object references in a distributed system.

Figure 4.1 Middleware layers

Applications, services

Middleware

layers

Underlying interprocess communication primitives:

UDP and TCP

This

chapter

Remote invocation, indirect communication

Sockets, message passing, multicast support, overlay networksSECTION 4.2 THE API FOR THE INTERNET PROTOCOLS 163

Section 4.4 discusses multicast communication: a form of interprocess

communication in which one process in a group of processes transmits the same

message to all members of the group. After explaining IP multicast, the section

discusses the need for more reliable forms of multicast.

Section 4.5 examines the increasingly important topic of overlay networks. An

overlay network is a network that is built over another network to permit applications to

route messages to destinations not specified by an IP address. Overlay networks can

enhance TCP/IP networks by providing alternative, more specialized network services.

They are important in supporting multicast communication and peer-to-peer

communication.

Finally, Section 4.6 presents a case study of a significant message-passing service,

MPI, developed by the high-performance computing community.

4.2 The API for the Internet protocols

In this section, we discuss the general characteristics of interprocess communication and

then discuss the Internet protocols as an example, explaining how programmers can use

them, either by means of UDP messages or through TCP streams.

Section 4.2.1 revisits the message communication operations send and receive

introduced in Section 2.3.2, with a discussion of how they synchronize with one another

and how message destinations are specified in a distributed system. Section 4.2.2

introduces sockets, which are used in the application programming interface to UDP and

TCP. Section 4.2.3 discusses UDP and its API in Java. Section 4.2.4 discusses TCP and

its API in Java. The APIs for Java are object oriented but are similar to the ones designed

originally in the Berkeley BSD 4.x UNIX operating system; a case study on the latter is

available on the web site for the book [www.cdk5.net/ipc]. Readers studying the

programming examples in this section should consult the online Java documentation or

Flanagan [2002] for the full specification of the classes discussed, which are in the

package java.net.

4.2.1 The characteristics of interprocess communication

Message passing between a pair of processes can be supported by two message

communication operations, send and receive, defined in terms of destinations and

messages. To communicate, one process sends a message (a sequence of bytes) to a

destination and another process at the destination receives the message. This activity

involves the communication of data from the sending process to the receiving process

and may involve the synchronization of the two processes. Section 4.2.3 gives

definitions for the send and receive operations in the Java API for the Internet protocols,

with a further case study of message passing (MPI) offered in Section 4.6.

Synchronous and asynchronous communication • A queue is associated with each

message destination. Sending processes cause messages to be added to remote queues and

receiving processes remove messages from local queues. Communication between the164 CHAPTER 4 INTERPROCESS COMMUNICATION

sending and receiving processes may be either synchronous or asynchronous. In the

synchronous form of communication, the sending and receiving processes synchronize at

every message. In this case, both send and receive are blocking operations. Whenever a

send is issued the sending process (or thread) is blocked until the corresponding receive is

issued. Whenever a receive is issued by a process (or thread), it blocks until a message

arrives.

In the asynchronous form of communication, the use of the send operation is nonblocking in that the sending process is allowed to proceed as soon as the message has

been copied to a local buffer, and the transmission of the message proceeds in parallel

with the sending process. The receive operation can have blocking and non-blocking

variants. In the non-blocking variant, the receiving process proceeds with its program

after issuing a receive operation, which provides a buffer to be filled in the background,

but it must separately receive notification that its buffer has been filled, by polling or

interrupt.

In a system environment such as Java, which supports multiple threads in a single

process, the blocking receive has no disadvantages, for it can be issued by one thread

while other threads in the process remain active, and the simplicity of synchronizing the

receiving threads with the incoming message is a substantial advantage. Non-blocking

communication appears to be more efficient, but it involves extra complexity in the

receiving process associated with the need to acquire the incoming message out of its

flow of control. For these reasons, today’s systems do not generally provide the nonblocking form of receive.

Message destinations • Chapter 3 explains that in the Internet protocols, messages are

sent to (Internet address, local port) pairs. A local port is a message destination within

a computer, specified as an integer. A port has exactly one receiver (multicast ports are

an exception, see Section 4.5.1) but can have many senders. Processes may use multiple

ports to receive messages. Any process that knows the number of a port can send a

message to it. Servers generally publicize their port numbers for use by clients.

If the client uses a fixed Internet address to refer to a service, then that service must

always run on the same computer for its address to remain valid. This can be avoided by

using the following approach to providing location transparency:

• Client programs refer to services by name and use a name server or binder (see

Section 5.4.2) to translate their names into server locations at runtime. This allows

services to be relocated but not to migrate – that is, to be moved while the system

is running.

Reliability • Chapter 2 defines reliable communication in terms of validity and

integrity. As far as the validity property is concerned, a point-to-point message service

can be described as reliable if messages are guaranteed to be delivered despite a

‘reasonable’ number of packets being dropped or lost. In contrast, a point-to-point

message service can be described as unreliable if messages are not guaranteed to be

delivered in the face of even a single packet dropped or lost. For integrity, messages

must arrive uncorrupted and without duplication.

Ordering • Some applications require that messages be delivered in sender order – that

is, the order in which they were transmitted by the sender. The delivery of messages out

of sender order is regarded as a failure by such applications.SECTION 4.2 THE API FOR THE INTERNET PROTOCOLS 165

4.2.2 Sockets

Both forms of communication (UDP and TCP) use the socket abstraction, which

provides an endpoint for communication between processes. Sockets originate from

BSD UNIX but are also present in most other versions of UNIX, including Linux as well

as Windows and the Macintosh OS. Interprocess communication consists of

transmitting a message between a socket in one process and a socket in another process,

as illustrated in Figure 4.2. For a process to receive messages, its socket must be bound

to a local port and one of the Internet addresses of the computer on which it runs.

Messages sent to a particular Internet address and port number can be received only by

a process whose socket is associated with that Internet address and port number.

Processes may use the same socket for sending and receiving messages. Each computer

has a large number (216) of possible port numbers for use by local processes for

receiving messages. Any process may make use of multiple ports to receive messages,

but a process cannot share ports with other processes on the same computer. (Processes

using IP multicast are an exception in that they do share ports – see Section 4.4.1.)

However, any number of processes may send messages to the same port. Each socket is

associated with a particular protocol – either UDP or TCP.

Java API for Internet addresses • As the IP packets underlying UDP and TCP are sent

to Internet addresses, Java provides a class, InetAddress, that represents Internet

addresses. Users of this class refer to computers by Domain Name System (DNS)

hostnames (see Section 3.4.7). For example, instances of InetAddress that contain

Internet addresses can be created by calling a static method of InetAddress, giving a

DNS hostname as the argument. The method uses the DNS to get the corresponding

Internet address. For example, to get an object representing the Internet address of the

host whose DNS name is bruno.dcs.qmul.ac.uk, use:

InetAddress aComputer = InetAddress.getByName("bruno.dcs.qmul.ac.uk");

This method can throw an UnknownHostException. Note that the user of the class does

not need to state the explicit value of an Internet address. In fact, the class encapsulates

the details of the representation of Internet addresses. Thus the interface for this class is

not dependent on the number of bytes needed to represent Internet addresses – 4 bytes

in IPv4 and 16 bytes in IPv6.

Figure 4.2 Sockets and ports

message

agreed port

socket any port socket

Internet address = 138.37.94.248 Internet address = 138.37.88.249

other ports

client server166 CHAPTER 4 INTERPROCESS COMMUNICATION

4.2.3 UDP datagram communication

A datagram sent by UDP is transmitted from a sending process to a receiving process

without acknowledgement or retries. If a failure occurs, the message may not arrive. A

datagram is transmitted between processes when one process sends it and another

receives it. To send or receive messages a process must first create a socket bound to an

Internet address of the local host and a local port. A server will bind its socket to a server

port – one that it makes known to clients so that they can send messages to it. A client

binds its socket to any free local port. The receive method returns the Internet address

and port of the sender, in addition to the message, allowing the recipient to send a reply.

The following are some issues relating to datagram communication:

Message size: The receiving process needs to specify an array of bytes of a particular

size in which to receive a message. If the message is too big for the array, it is

truncated on arrival. The underlying IP protocol allows packet lengths of up to 216

bytes, which includes the headers as well as the message. However, most

environments impose a size restriction of 8 kilobytes. Any application requiring

messages larger than the maximum must fragment them into chunks of that size.

Generally, an application, for example DNS, will decide on a size that is not

excessively large but is adequate for its intended use.

Blocking: Sockets normally provide non-blocking sends and blocking receives for

datagram communication (a non-blocking receive is an option in some

implementations). The send operation returns when it has handed the message to the

underlying UDP and IP protocols, which are responsible for transmitting it to its

destination. On arrival, the message is placed in a queue for the socket that is bound

to the destination port. The message can be collected from the queue by an

outstanding or future invocation of receive on that socket. Messages are discarded at

the destination if no process already has a socket bound to the destination port.

The method receive blocks until a datagram is received, unless a timeout has

been set on the socket. If the process that invokes the receive method has other work

to do while waiting for the message, it should arrange to use a separate thread.

Threads are discussed in Chapter 7. For example, when a server receives a message

from a client, the message may specify work to do, in which case the server will use

separate threads to do the work and to wait for messages from other clients.

Timeouts: The receive that blocks forever is suitable for use by a server that is waiting

to receive requests from its clients. But in some programs, it is not appropriate that a

process that has invoked a receive operation should wait indefinitely in situations

where the sending process may have crashed or the expected message may have been

lost. To allow for such requirements, timeouts can be set on sockets. Choosing an

appropriate timeout interval is difficult, but it should be fairly large in comparison

with the time required to transmit a message.

Receive from any: The receive method does not specify an origin for messages.

Instead, an invocation of receive gets a message addressed to its socket from any

origin. The receive method returns the Internet address and local port of the sender,

allowing the recipient to check where the message came from. It is possible to

connect a datagram socket to a particular remote port and Internet address, in which

case the socket is only able to send messages to and receive messages from that

address.SECTION 4.2 THE API FOR THE INTERNET PROTOCOLS 167

Failure model for UDP datagrams • Chapter 2 presents a failure model for

communication channels and defines reliable communication in terms of two properties:

integrity and validity. The integrity property requires that messages should not be

corrupted or duplicated. The use of a checksum ensures that there is a negligible

probability that any message received is corrupted. UDP datagrams suffer from the

following failures:

Omission failures: Messages may be dropped occasionally, either because of a

checksum error or because no buffer space is available at the source or destination.

To simplify the discussion, we regard send-omission and receive-omission failures

(see Figure 2.15) as omission failures in the communication channel.

Ordering: Messages can sometimes be delivered out of sender order.

Applications using UDP datagrams are left to provide their own checks to achieve the

quality of reliable communication they require. A reliable delivery service may be

constructed from one that suffers from omission failures by the use of

acknowledgements. Section 5.2 discusses how reliable request-reply protocols for

client-server communication may be built over UDP.

Use of UDP • For some applications, it is acceptable to use a service that is liable to

occasional omission failures. For example, the Domain Name System, which looks up

DNS names in the Internet, is implemented over UDP. Voice over IP (VOIP) also runs

over UDP. UDP datagrams are sometimes an attractive choice because they do not

suffer from the overheads associated with guaranteed message delivery. There are three

main sources of overhead:

• the need to store state information at the source and destination;

• the transmission of extra messages;

• latency for the sender.

The reasons for these overheads are discussed in Section 4.2.4.

Java API for UDP datagrams • The Java API provides datagram communication by

means of two classes: DatagramPacket and DatagramSocket.

DatagramPacket: This class provides a constructor that makes an instance out of an

array of bytes comprising a message, the length of the message and the Internet

address and local port number of the destination socket, as follows:

An instance of DatagramPacket may be transmitted between processes when one

process sends it and another receives it.

This class provides another constructor for use when receiving a message. Its

arguments specify an array of bytes in which to receive the message and the length

of the array. A received message is put in the DatagramPacket together with its

length and the Internet address and port of the sending socket. The message can be

retrieved from the DatagramPacket by means of the method getData. The methods

getPort and getAddress access the port and Internet address.

Datagram packet

array of bytes containing message length of message Internet address port number168 CHAPTER 4 INTERPROCESS COMMUNICATION

DatagramSocket: This class supports sockets for sending and receiving UDP

datagrams. It provides a constructor that takes a port number as its argument, for use

by processes that need to use a particular port. It also provides a no-argument

constructor that allows the system to choose a free local port. These constructors can

throw a SocketException if the chosen port is already in use or if a reserved port (a

number below 1024) is specified when running over UNIX.

The class DatagramSocket provides methods that include the following:

send and receive: These methods are for transmitting datagrams between a pair

of sockets. The argument of send is an instance of DatagramPacket containing

a message and its destination. The argument of receive is an empty

DatagramPacket in which to put the message, its length and its origin. The

methods send and receive can throw IOExceptions.

setSoTimeout: This method allows a timeout to be set. With a timeout set, the receive

method will block for the time specified and then throw an InterruptedIOException.

connect: This method is used for connecting to a particular remote port and

Internet address, in which case the socket is only able to send messages to and

receive messages from that address.

Figure 4.3 UDP client sends a message to the server and gets a reply

import java.net.\*;

import java.io.\*;

public class UDPClient{

public static void main(String args[]){

// args give message contents and server hostname

DatagramSocket aSocket = null;

try {

aSocket = new DatagramSocket();

byte [] m = args[0].getBytes();

InetAddress aHost = InetAddress.getByName(args[1]);

int serverPort = 6789;

DatagramPacket request =

new DatagramPacket(m, m.length(), aHost, serverPort);

aSocket.send(request);

byte[] buffer = new byte[1000];

DatagramPacket reply = new DatagramPacket(buffer, buffer.length);

aSocket.receive(reply);

System.out.println("Reply: " + new String(reply.getData()));

} catch (SocketException e){System.out.println("Socket: " + e.getMessage());

} catch (IOException e){System.out.println("IO: " + e.getMessage());

} finally { if(aSocket != null) aSocket.close();}

}

}SECTION 4.2 THE API FOR THE INTERNET PROTOCOLS 169

Figure 4.3 shows the program for a client that creates a socket, sends a message to a

server at port 6789 and then waits to receive a reply. The arguments of the main method

supply a message and the DNS hostname of the server. The message is converted to an

array of bytes, and the DNS hostname is converted to an Internet address. Figure 4.4

shows the program for the corresponding server, which creates a socket bound to its

server port (6789) and then repeatedly waits to receive a request message from a client,

to which it replies by sending back the same message.

4.2.4 TCP stream communication

The API to the TCP protocol, which originates from BSD 4.x UNIX, provides the

abstraction of a stream of bytes to which data may be written and from which data may

be read. The following characteristics of the network are hidden by the stream

abstraction:

Message sizes: The application can choose how much data it writes to a stream or

reads from it. It may deal in very small or very large sets of data. The underlying

implementation of a TCP stream decides how much data to collect before

transmitting it as one or more IP packets. On arrival, the data is handed to the

application as requested. Applications can, if necessary, force data to be sent

immediately.

Lost messages: The TCP protocol uses an acknowledgement scheme. As an example

of a simple scheme (which is not used in TCP), the sending end keeps a record of each

Figure 4.4 UDP server repeatedly receives a request and sends it back to the client

import java.net.\*;

import java.io.\*;

public class UDPServer{

public static void main(String args[]){

DatagramSocket aSocket = null;

try{

aSocket = new DatagramSocket(6789);

byte[] buffer = new byte[1000];

while(true){

DatagramPacket request = new DatagramPacket(buffer, buffer.length);

aSocket.receive(request);

DatagramPacket reply = new DatagramPacket(request.getData(),

request.getLength(), request.getAddress(), request.getPort());

aSocket.send(reply);

}

} catch (SocketException e){System.out.println("Socket: " + e.getMessage());

} catch (IOException e) {System.out.println("IO: " + e.getMessage());

} finally {if (aSocket != null) aSocket.close();}

}

}170 CHAPTER 4 INTERPROCESS COMMUNICATION

IP packet sent and the receiving end acknowledges all the arrivals. If the sender does

not receive an acknowledgement within a timeout, it retransmits the message. The

more sophisticated sliding window scheme [Comer 2006] cuts down on the number

of acknowledgement messages required.

Flow control: The TCP protocol attempts to match the speeds of the processes that

read from and write to a stream. If the writer is too fast for the reader, then it is

blocked until the reader has consumed sufficient data.

Message duplication and ordering: Message identifiers are associated with each IP

packet, which enables the recipient to detect and reject duplicates, or to reorder

messages that do not arrive in sender order.

Message destinations: A pair of communicating processes establish a connection

before they can communicate over a stream. Once a connection is established, the

processes simply read from and write to the stream without needing to use Internet

addresses and ports. Establishing a connection involves a connect request from client

to server followed by an accept request from server to client before any

communication can take place. This could be a considerable overhead for a single

client-server request and reply.

The API for stream communication assumes that when a pair of processes are

establishing a connection, one of them plays the client role and the other plays the server

role, but thereafter they could be peers. The client role involves creating a stream socket

bound to any port and then making a connect request asking for a connection to a server

at its server port. The server role involves creating a listening socket bound to a server

port and waiting for clients to request connections. The listening socket maintains a

queue of incoming connection requests. In the socket model, when the server accepts a

connection, a new stream socket is created for the server to communicate with a client,

meanwhile retaining its socket at the server port for listening for connect requests from

other clients.

The pair of sockets in the client and server are connected by a pair of streams, one

in each direction. Thus each socket has an input stream and an output stream. One of the

pair of processes can send information to the other by writing to its output stream, and

the other process obtains the information by reading from its input stream.

When an application closes a socket, this indicates that it will not write any more

data to its output stream. Any data in the output buffer is sent to the other end of the

stream and put in the queue at the destination socket, with an indication that the stream

is broken. The process at the destination can read the data in the queue, but any further

reads after the queue is empty will result in an indication of end of stream. When a

process exits or fails, all of its sockets are eventually closed and any process attempting

to communicate with it will discover that its connection has been broken.

The following are some outstanding issues related to stream communication:

Matching of data items: Two communicating processes need to agree as to the

contents of the data transmitted over a stream. For example, if one process writes an

int followed by a double to a stream, then the reader at the other end must read an int

followed by a double. When a pair of processes do not cooperate correctly in their

use of a stream, the reading process may experience errors when interpreting the data

or may block due to insufficient data in the stream.SECTION 4.2 THE API FOR THE INTERNET PROTOCOLS 171

Blocking: The data written to a stream is kept in a queue at the destination socket.

When a process attempts to read data from an input channel, it will get data from the

queue or it will block until data becomes available. The process that writes data to a

stream may be blocked by the TCP flow-control mechanism if the socket at the other

end is queuing as much data as the protocol allows.

Threads: When a server accepts a connection, it generally creates a new thread in

which to communicate with the new client. The advantage of using a separate thread

for each client is that the server can block when waiting for input without delaying

other clients. In an environment in which threads are not provided, an alternative is

to test whether input is available from a stream before attempting to read it; for

example, in a UNIX environment the select system call may be used for this purpose.

Failure model • To satisfy the integrity property of reliable communication, TCP

streams use checksums to detect and reject corrupt packets and sequence numbers to

detect and reject duplicate packets. For the sake of the validity property, TCP streams

use timeouts and retransmissions to deal with lost packets. Therefore, messages are

guaranteed to be delivered even when some of the underlying packets are lost.

But if the packet loss over a connection passes some limit or the network

connecting a pair of communicating processes is severed or becomes severely

congested, the TCP software responsible for sending messages will receive no

acknowledgements and after a time will declare the connection to be broken. Thus TCP

does not provide reliable communication, because it does not guarantee to deliver

messages in the face of all possible difficulties.

When a connection is broken, a process using it will be notified if it attempts to

read or write. This has the following effects:

• The processes using the connection cannot distinguish between network failure

and failure of the process at the other end of the connection.

• The communicating processes cannot tell whether the messages they have sent

recently have been received or not.

Use of TCP • Many frequently used services run over TCP connections, with reserved

port numbers. These include the following:

HTTP: The Hypertext Transfer Protocol is used for communication between web

browsers and web servers; it is discussed in Section 5.2.

FTP: The File Transfer Protocol allows directories on a remote computer to be

browsed and files to be transferred from one computer to another over a connection.

Telnet: Telnet provides access by means of a terminal session to a remote computer.

SMTP: The Simple Mail Transfer Protocol is used to send mail between computers.

Java API for TCP streams • The Java interface to TCP streams is provided in the classes

ServerSocket and Socket:

ServerSocket: This class is intended for use by a server to create a socket at a server

port for listening for connect requests from clients. Its accept method gets a connect

request from the queue or, if the queue is empty, blocks until one arrives. The result

of executing accept is an instance of Socket – a socket to use for communicating with

the client.172 CHAPTER 4 INTERPROCESS COMMUNICATION

Socket: This class is for use by a pair of processes with a connection. The client uses

a constructor to create a socket, specifying the DNS hostname and port of a server.

This constructor not only creates a socket associated with a local port but also

connects it to the specified remote computer and port number. It can throw an

UnknownHostException if the hostname is wrong or an IOException if an IO error

occurs.

The Socket class provides the methods getInputStream and getOutputStream

for accessing the two streams associated with a socket. The return types of these

methods are InputStream and OutputStream, respectively – abstract classes that

define methods for reading and writing bytes. The return values can be used as the

arguments of constructors for suitable input and output streams. Our example uses

DataInputStream and DataOutputStream, which allow binary representations of

primitive data types to be read and written in a machine-independent manner.

Figure 4.5 shows a client program in which the arguments of the main method supply a

message and the DNS hostname of the server. The client creates a socket bound to the

hostname and server port 7896. It makes a DataInputStream and a DataOutputStream

from the socket’s input and output streams, then writes the message to its output stream

and waits to read a reply from its input stream. The server program in Figure 4.6 opens

a server socket on its server port (7896) and listens for connect requests. When one

arrives, it makes a new thread in which to communicate with the client. The new thread

Figure 4.5 TCP client makes connection to server, sends request and receives reply

import java.net.\*;

import java.io.\*;

public class TCPClient {

public static void main (String args[]) {

// arguments supply message and hostname of destination

Socket s = null;

try{

int serverPort = 7896;

s = new Socket(args[1], serverPort);

DataInputStream in = new DataInputStream( s.getInputStream());

DataOutputStream out =

new DataOutputStream( s.getOutputStream());

out.writeUTF(args[0]); // UTF is a string encoding; see Sec 4.3

String data = in.readUTF();

System.out.println("Received: "+ data) ;

}catch (UnknownHostException e){

System.out.println("Sock:"+e.getMessage());

} catch (EOFException e){System.out.println("EOF:"+e.getMessage());

} catch (IOException e){System.out.println("IO:"+e.getMessage());

} finally {if(s!=null) try {s.close();}catch (IOException e){/\*close failed\*/}}

}

}SECTION 4.2 THE API FOR THE INTERNET PROTOCOLS 173

creates a DataInputStream and a DataOutputStream from its socket’s input and output

streams and then waits to read a message and write the same one back.

As our message consists of a string, the client and server processes use the method

writeUTF of DataOutputStream to write it to the output stream and the method

readUTF of DataInputStream to read it from the input stream. UTF-8 is an encoding that

represents strings in a particular format, which is described in Section 4.3.

Figure 4.6 TCP server makes a connection for each client and then echoes the client’s request

import java.net.\*;

import java.io.\*;

public class TCPServer {

public static void main (String args[]) {

try{

int serverPort = 7896;

ServerSocket listenSocket = new ServerSocket(serverPort);

while(true) {

Socket clientSocket = listenSocket.accept();

Connection c = new Connection(clientSocket);

}

} catch(IOException e) {System.out.println("Listen :"+e.getMessage());}

}

}

class Connection extends Thread {

DataInputStream in;

DataOutputStream out;

Socket clientSocket;

public Connection (Socket aClientSocket) {

try {

clientSocket = aClientSocket;

in = new DataInputStream( clientSocket.getInputStream());

out =new DataOutputStream( clientSocket.getOutputStream());

this.start();

} catch(IOException e) {System.out.println("Connection:"+e.getMessage());}

}

public void run(){

try { // an echo server

String data = in.readUTF();

out.writeUTF(data);

} catch(EOFException e) {System.out.println("EOF:"+e.getMessage());

} catch(IOException e) {System.out.println("IO:"+e.getMessage());

} finally { try {clientSocket.close();}catch (IOException e){/\*close failed\*/}}

}

}174 CHAPTER 4 INTERPROCESS COMMUNICATION

When a process has closed its socket, it will no longer be able to use its input and

output streams. The process to which it has sent data can read the data in its queue, but

any further reads after the queue is empty will result in an EOFException. Attempts to

use a closed socket or to write to a broken stream result in an IOException.

4.3 External data representation and marshalling

The information stored in running programs is represented as data structures – for

example, by sets of interconnected objects – whereas the information in messages

consists of sequences of bytes. Irrespective of the form of communication used, the data

structures must be flattened (converted to a sequence of bytes) before transmission and

rebuilt on arrival. The individual primitive data items transmitted in messages can be

data values of many different types, and not all computers store primitive values such as

integers in the same order. The representation of floating-point numbers also differs

between architectures. There are two variants for the ordering of integers: the so-called

big-endian order, in which the most significant byte comes first; and little-endian order,

in which it comes last. Another issue is the set of codes used to represent characters: for

example, the majority of applications on systems such as UNIX use ASCII character

coding, taking one byte per character, whereas the Unicode standard allows for the

representation of texts in many different languages and takes two bytes per character.

One of the following methods can be used to enable any two computers to

exchange binary data values:

• The values are converted to an agreed external format before transmission and

converted to the local form on receipt; if the two computers are known to be the

same type, the conversion to external format can be omitted.

• The values are transmitted in the sender’s format, together with an indication of

the format used, and the recipient converts the values if necessary.

Note, however, that bytes themselves are never altered during transmission. To support

RMI or RPC, any data type that can be passed as an argument or returned as a result must

be able to be flattened and the individual primitive data values represented in an agreed

format. An agreed standard for the representation of data structures and primitive values

is called an external data representation.

Marshalling is the process of taking a collection of data items and assembling

them into a form suitable for transmission in a message. Unmarshalling is the process

of disassembling them on arrival to produce an equivalent collection of data items at the

destination. Thus marshalling consists of the translation of structured data items and

primitive values into an external data representation. Similarly, unmarshalling consists

of the generation of primitive values from their external data representation and the

rebuilding of the data structures.

Three alternative approaches to external data representation and marshalling are

discussed (with a fourth considered in Chapter 21, when we examine Google’s approach

to representing structured data):SECTION 4.3 EXTERNAL DATA REPRESENTATION AND MARSHALLING 175

• CORBA’s common data representation, which is concerned with an external

representation for the structured and primitive types that can be passed as the

arguments and results of remote method invocations in CORBA. It can be used by

a variety of programming languages (see Chapter 8).

• Java’s object serialization, which is concerned with the flattening and external

data representation of any single object or tree of objects that may need to be

transmitted in a message or stored on a disk. It is for use only by Java.

• XML (Extensible Markup Language), which defines a textual fomat for

representing structured data. It was originally intended for documents containing

textual self-describing structured data – for example documents accessible on the

Web – but it is now also used to represent the data sent in messages exchanged by

clients and servers in web services (see Chapter 9).

In the first two cases, the marshalling and unmarshalling activities are intended to be

carried out by a middleware layer without any involvement on the part of the application

programmer. Even in the case of XML, which is textual and therefore more accessible

to hand-encoding, software for marshalling and unmarshalling is available for all

commonly used platforms and programming environments. Because marshalling

requires the consideration of all the finest details of the representation of the primitive

components of composite objects, the process is likely to be error-prone if carried out

by hand. Compactness is another issue that can be addressed in the design of

automatically generated marshalling procedures.

In the first two approaches, the primitive data types are marshalled into a binary

form. In the third approach (XML), the primitive data types are represented textually.

The textual representation of a data value will generally be longer than the equivalent

binary representation. The HTTP protocol, which is described in Chapter 5, is another

example of the textual approach.

Another issue with regard to the design of marshalling methods is whether the

marshalled data should include information concerning the type of its contents. For

example, CORBA’s representation includes just the values of the objects transmitted,

and nothing about their types. On the other hand, both Java serialization and XML do

include type information, but in different ways. Java puts all of the required type

information into the serialized form, but XML documents may refer to externally

defined sets of names (with types) called namespaces.

Although we are interested in the use of an external data representation for the

arguments and results of RMIs and RPCs, it does have a more general use for

representing data structures, objects or structured documents in a form suitable for

transmission in messages or storing in files.

Two other techniques for external data representation are worthy of mention.

Google uses an approach called protocol buffers to capture representations of both

stored and transmitted data, which we examine in Section 20.4.1. There is also

considerable interest in JSON (JavaScript Object Notation) as an approach to external

data representation [www.json.org]. Protocol buffers and JSON represent a step towards

more lightweight approaches to data representation (when compared, for example, to

XML).176 CHAPTER 4 INTERPROCESS COMMUNICATION

4.3.1 CORBA’s Common Data Representation (CDR)

CORBA CDR is the external data representation defined with CORBA 2.0 [OMG

2004a]. CDR can represent all of the data types that can be used as arguments and return

values in remote invocations in CORBA. These consist of 15 primitive types, which

include short (16-bit), long (32-bit), unsigned short, unsigned long, float (32-bit),

double (64-bit), char, boolean (TRUE, FALSE), octet (8-bit), and any (which can

represent any basic or constructed type); together with a range of composite types,

which are described in Figure 4.7. Each argument or result in a remote invocation is

represented by a sequence of bytes in the invocation or result message.

Primitive types: CDR defines a representation for both big-endian and little-endian

orderings. Values are transmitted in the sender’s ordering, which is specified in each

message. The recipient translates if it requires a different ordering. For example, a

16-bit short occupies two bytes in the message, and for big-endian ordering, the most

significant bits occupy the first byte and the least significant bits occupy the second

byte. Each primitive value is placed at an index in the sequence of bytes according to

its size. Suppose that the sequence of bytes is indexed from zero upwards. Then a

primitive value of size n bytes (where n = 1, 2, 4 or 8) is appended to the sequence at

an index that is a multiple of n in the stream of bytes. Floating-point values follow

the IEEE standard, in which the sign, exponent and fractional part are in bytes 0–n

for big-endian ordering and the other way round for little-endian. Characters are

represented by a code set agreed between client and server.

Constructed types: The primitive values that comprise each constructed type are

added to a sequence of bytes in a particular order, as shown in Figure 4.7.

Figure 4.8 shows a message in CORBA CDR that contains the three fields of a struct

whose respective types are string, string and unsigned long. The figure shows the

sequence of bytes with four bytes in each row. The representation of each string consists

of an unsigned long representing its length followed by the characters in the string. For

simplicity, we assume that each character occupies just one byte. Variable-length data

is padded with zeros so that it has a standard form, enabling marshalled data or its

checksum to be compared. Note that each unsigned long, which occupies four bytes,

Figure 4.7 CORBA CDR for constructed types

Type Representation

sequence length (unsigned long) followed by elements in order

string length (unsigned long) followed by characters in order (can also

have wide characters)

array array elements in order (no length specified because it is fixed)

struct in the order of declaration of the components

enumerated unsigned long (the values are specified by the order declared)

union type tag followed by the selected memberSECTION 4.3 EXTERNAL DATA REPRESENTATION AND MARSHALLING 177

starts at an index that is a multiple of four. The figure does not distinguish between the

big- and little-endian orderings. Although the example in Figure 4.8 is simple, CORBA

CDR can represent any data structure that can be composed from the primitive and

constructed types, but without using pointers.

Another example of an external data representation is the Sun XDR standard,

which is specified in RFC 1832 [Srinivasan 1995b] and described in www.cdk5.net/ipc.

It was developed by Sun for use in the messages exchanged between clients and servers

in Sun NFS (see Chapter 13).

The type of a data item is not given with the data representation in the message in

either the CORBA CDR or the Sun XDR standard. This is because it is assumed that the

sender and recipient have common knowledge of the order and types of the data items

in a message. In particular, for RMI or RPC, each method invocation passes arguments

of particular types, and the result is a value of a particular type.

Marshalling in CORBA • Marshalling operations can be generated automatically from

the specification of the types of data items to be transmitted in a message. The types of

the data structures and the types of the basic data items are described in CORBA IDL

(see Section 8.3.1), which provides a notation for describing the types of the arguments

and results of RMI methods. For example, we might use CORBA IDL to describe the

data structure in the message in Figure 4.8 as follows:

struct Person{

string name;

string place;

unsigned long year;

};

The CORBA interface compiler (see Chapter 5) generates appropriate marshalling and

unmarshalling operations for the arguments and results of remote methods from the

definitions of the types of their parameters and results.

Figure 4.8 CORBA CDR message

index in

sequence of bytes

notes

on representation

0–3 5 length of string

4–7 "Smit" ‘Smith’

8–11 "h\_\_\_"

12–15 6 length of string

16–19 "Lond" ‘London’

20–23 "on\_\_"

24–27 1984 unsigned long

4 bytes

The flattened form represents a Person struct with value: {‘Smith’, ‘London’, 1984}178 CHAPTER 4 INTERPROCESS COMMUNICATION

4.3.2 Java object serialization

In Java RMI, both objects and primitive data values may be passed as arguments and

results of method invocations. An object is an instance of a Java class. For example, the

Java class equivalent to the Person struct defined in CORBA IDL might be:

public class Person implements Serializable {

private String name;

private String place;

private int year;

public Person(String aName, String aPlace, int aYear) {

name = aName;

place = aPlace;

year = aYear;

}

// followed by methods for accessing the instance variables

}

The above class states that it implements the Serializable interface, which has no

methods. Stating that a class implements the Serializable interface (which is provided in

the java.io package) has the effect of allowing its instances to be serialized.

In Java, the term serialization refers to the activity of flattening an object or a

connected set of objects into a serial form that is suitable for storing on disk or

transmitting in a message, for example, as an argument or the result of an RMI.

Deserialization consists of restoring the state of an object or a set of objects from their

serialized form. It is assumed that the process that does the deserialization has no prior

knowledge of the types of the objects in the serialized form. Therefore some information

about the class of each object is included in the serialized form. This information enables

the recipient to load the appropriate class when an object is deserialized.

The information about a class consists of the name of the class and a version

number. The version number is intended to change when major changes are made to the

class. It can be set by the programmer or calculated automatically as a hash of the name

of the class and its instance variables, methods and interfaces. The process that

deserializes an object can check that it has the correct version of the class.

Java objects can contain references to other objects. When an object is serialized,

all the objects that it references are serialized together with it to ensure that when the

object is reconstructed, all of its references can be fulfilled at the destination. References

are serialized as handles. In this case, the handle is a reference to an object within the

serialized form – for example, the next number in a sequence of positive integers. The

serialization procedure must ensure that there is a 1–1 correspondence between object

references and handles. It must also ensure that each object is written once only – on the

second or subsequent occurrence of an object, the handle is written instead of the object.

To serialize an object, its class information is written out, followed by the types

and names of its instance variables. If the instance variables belong to new classes, then

their class information must also be written out, followed by the types and names of their

instance variables. This recursive procedure continues until the class information and

types and names of the instance variables of all of the necessary classes have beenSECTION 4.3 EXTERNAL DATA REPRESENTATION AND MARSHALLING 179

written out. Each class is given a handle, and no class is written more than once to the

stream of bytes (the handles being written instead where necessary).

The contents of the instance variables that are primitive types, such as integers,

chars, booleans, bytes and longs, are written in a portable binary format using methods

of the ObjectOutputStream class. Strings and characters are written by its writeUTF

method using the Universal Transfer Format (UTF-8), which enables ASCII characters

to be represented unchanged (in one byte), whereas Unicode characters are represented

by multiple bytes. Strings are preceded by the number of bytes they occupy in the

stream.

As an example, consider the serialization of the following object:

Person p = new Person("Smith", "London", 1984);

The serialized form is illustrated in Figure 4.9, which omits the values of the handles and

of the type markers that indicate the objects, classes, strings and other objects in the full

serialized form. The first instance variable (1984) is an integer that has a fixed length;

the second and third instance variables are strings and are preceded by their lengths.

To make use of Java serialization, for example to serialize the Person object,

create an instance of the class ObjectOutputStream and invoke its writeObject method,

passing the Person object as its argument. To deserialize an object from a stream of data,

open an ObjectInputStream on the stream and use its readObject method to reconstruct

the original object. The use of this pair of classes is similar to the use of

DataOutputStream and DataInputStream, illustrated in Figures 4.5 and 4.6.

Serialization and deserialization of the arguments and results of remote

invocations are generally carried out automatically by the middleware, without any

participation by the application programmer. If necessary, programmers with special

requirements may write their own version of the methods that read and write objects. To

find out how to do this and to get further information about serialization in Java, read

the tutorial on object serialization [java.sun.com II]. Another way in which a

programmer may modify the effects of serialization is by declaring variables that should

not be serialized as transient. Examples of things that should not be serialized are

references to local resources such as files and sockets.

The use of reflection • The Java language supports reflection – the ability to enquire

about the properties of a class, such as the names and types of its instance variables and

methods. It also enables classes to be created from their names, and a constructor with

Figure 4.9 Indication of Java serialized form

Serialized values Explanation

Person 8-byte version number h0 class name, version number

3 int year java.lang.String

name

java.lang.String

place

number, type and name of

instance variables

1984 5 Smith 6 London h1 values of instance variables

The true serialized form contains additional type markers; h0 and h1 are handles.180 CHAPTER 4 INTERPROCESS COMMUNICATION

given argument types to be created for a given class. Reflection makes it possible to do

serialization and deserialization in a completely generic manner. This means that there

is no need to generate special marshalling functions for each type of object, as described

above for CORBA. To find out more about reflection, see Flanagan [2002].

Java object serialization uses reflection to find out the class name of the object to

be serialized and the names, types and values of its instance variables. That is all that is

needed for the serialized form.

For deserialization, the class name in the serialized form is used to create a class.

This is then used to create a new constructor with argument types corresponding to those

specified in the serialized form. Finally, the new constructor is used to create a new

object with instance variables whose values are read from the serialized form.

4.3.3 Extensible Markup Language (XML)

XML is a markup language that was defined by the World Wide Web Consortium

(W3C) for general use on the Web. In general, the term markup language refers to a

textual encoding that represents both a text and details as to its structure or its

appearance. Both XML and HTML were derived from SGML (Standardized

Generalized Markup Language) [ISO 8879], a very complex markup language. HTML

(see Section 1.6) was designed for defining the appearance of web pages. XML was

designed for writing structured documents for the Web.

XML data items are tagged with ‘markup’ strings. The tags are used to describe

the logical structure of the data and to associate attribute-value pairs with logical

structures. That is, in XML, the tags relate to the structure of the text that they enclose,

in contrast to HTML, in which the tags specify how a browser could display the text. For

a specification of XML, see the pages on XML provided by W3C [www.w3.org VI].

XML is used to enable clients to communicate with web services and for defining

the interfaces and other properties of web services. However, XML is also used in many

other ways, including in archiving and retrieval systems – although an XML archive

may be larger than a binary one, it has the advantage of being readable on any computer.

Other examples of uses of XML include for the specification of user interfaces and the

encoding of configuration files in operating systems.

XML is extensible in the sense that users can define their own tags, in contrast to

HTML, which uses a fixed set of tags. However, if an XML document is intended to be

used by more than one application, then the names of the tags must be agreed between

them. For example, clients usually use SOAP messages to communicate with web

services. SOAP (see Section 9.2.1) is an XML format whose tags are published for use

by web services and their clients.

Some external data representations (such as CORBA CDR) do not need to be selfdescribing, because it is assumed that the client and server exchanging a message have

prior knowledge of the order and the types of the information it contains. However,

XML was intended to be used by multiple applications for different purposes. The

provision of tags, together with the use of namespaces to define the meaning of the tags,

has made this possible. In addition, the use of tags enables applications to select just

those parts of a document it needs to process: it will not be affected by the addition of

information relevant to other applications.SECTION 4.3 EXTERNAL DATA REPRESENTATION AND MARSHALLING 181

XML documents, being textual, can be read by humans. In practice, most XML

documents are generated and read by XML processing software, but the ability to read

XML can be useful when things go wrong. In addition, the use of text makes XML

independent of any particular platform. The use of a textual rather than a binary

representation, together with the use of tags, makes the messages large, so they require

longer processing and transmission times, as well as more space to store. A comparison

of the efficiency of messages using the SOAP XML format and CORBA CDR is given

in Section 9.2.4. However, files and messages can be compressed – HTTP version 1.1

allows data to be compressed, which saves bandwidth during transmission.

XML elements and attributes • Figure 4.10 shows the XML definition of the Person

structure that was used to illustrate marshalling in CORBA CDR and Java. It shows that

XML consists of tags and character data. The character data, for example Smith or 1984,

is the actual data. As in HTML, the structure of an XML document is defined by pairs

of tags enclosed in angle brackets. In Figure 4.10, <name> and <place> are both tags.

As in HTML, layout can generally be used to improve readability. Comments in XML

are denoted in the same way as those in HTML.

Elements: An element in XML consists of a portion of character data surrounded by

matching start and end tags. For example, one of the elements in Figure 4.10 consists of

the data Smith contained within the <name> ... </name> tag pair. Note that the element

with the <name> tag is enclosed in the element with the <person id="123456789"> ...

</person > tag pair. The ability of an element to enclose another element allows

hierarchic data to be represented – a very important aspect of XML. An empty tag has

no content and is terminated with /> instead of >. For example, the empty tag

<european/> could be included within the <person> ...</person> tag.

Attributes: A start tag may optionally include pairs of associated attribute names and

values such as id="123456789", as shown above. The syntax is the same as for HTML,

in which an attribute name is followed by an equal sign and an attribute value in quotes.

Multiple attribute values are separated by spaces.

It is a matter of choice as to which items are represented as elements and which

ones as attributes. An element is generally a container for data, whereas an attribute is

used for labelling that data. In our example, 123456789 might be an identifier used by

the application, whereas name, place and year might be displayed. Also, if data contains

substructures or several lines, it must be defined as an element. Attributes are for simple

values.

Figure 4.10 XML definition of the Person structure

<person id="123456789">

<name>Smith</name>

<place>London</place>

<year>1984</year>

<!-- a comment -->

</person >182 CHAPTER 4 INTERPROCESS COMMUNICATION

Names: The names of tags and attributes in XML generally start with a letter, but can

also start with an underline or a colon. The names continue with letters, digits, hyphens,

underscores, colons or full stops. Letters are case-sensitive. Names that start with xml

are reserved.

Binary data: All of the information in XML elements must be expressed as character

data. But the question is: how do we represent encrypted elements or secure hashes –

both of which, as we shall see in Section 9.5 are used in XML security? The answer is

that they can be represented in base64 notation [Freed and Borenstein 1996], which uses

only the alphanumeric characters together with +, / and =, which has a special meaning.

Parsing and well-formed documents • An XML document must be well formed – that

is, it must conform to rules about its structure. A basic rule is that every start tag has a

matching end tag. Another basic rule is that all tags are correctly nested – for example,

<x>..<y>..</y>..</x> is correct, whereas <x>..<y>..</x>..</y> is not. Finally, every

XML document must have a single root element that encloses all the other elements.

These rules make it very simple to implement parsers for XML documents. When a

parser reads an XML document that is not well formed, it will report a fatal error.

CDATA: XML parsers normally parse the contents of elements because they may contain

further nested structures. If text needs to contain an angle bracket or a quote, it may be

represented in a special way: for example, &lt represents the opening angle bracket.

However, if a section should not be parsed – for example, because it contains special

characters – it can be denoted as CDATA. For example, if a place name is to include an

apostrophe, then it could be specified in either of the two following ways:

<place> King&apos Cross </place >

<place> <![CDATA [King's Cross]]></place >

XML prolog: Every XML document must have a prolog as its first line. The prolog must

at least specify the version of XML in use (which is currently 1.0). For example:

<?XML version = "1.0" encoding = "UTF-8" standalone = "yes"?>

The prolog may specify the encoding (UTF-8 is the default and was explained in Section

4.3.2). The term encoding refers to the set of codes used to represent characters – ASCII

being the best-known example. Note that in the XML prolog, ASCII is specified as usascii. Other possible encodings include ISO-8859-1 (or Latin-1) – an 8-bit encoding

whose first 128 values are ASCII, with the rest being used to represent the characters in

Western European languages – and various other 8-bit encodings for representing other

alphabets, for example, Greek or Cyrillic.

An additional attribute may be used to state whether the document stands alone or

is dependent on external definitions.

XML namespaces • Traditionally, namespaces provide a means for scoping names. An

XML namespace is a set of names for a collection of element types and attributes that is

referenced by a URL. Any other XML document can use an XML namespace by

referring to its URL.

Any element that makes use of an XML namespace can specify that namespace as

an attribute called xmlns, whose value is a URL referring to the file containing the

namespace definitions. For example:

xmlns:pers = "http://www.cdk5.net/person"SECTION 4.3 EXTERNAL DATA REPRESENTATION AND MARSHALLING 183

The name after xmlns, in this case pers can be used as a prefix to refer to the elements

in a particular namespace, as shown in Figure 4.11. The pers prefix is bound to

http://www.cdk4.net/person for the person element. A namespace applies within the

context of the enclosing pair of start and end tags unless overridden by an enclosed

namespace declaration. An XML document may be defined in terms of several different

namespaces, each of which is referenced by a unique prefix.

The namespace convention allows an application to make use of multiple sets of

external definitions in different namespaces without the risk of name clashes.

XML schemas • An XML schema [www.w3.org VIII] defines the elements and

attributes that can appear in a document, how the elements are nested and the order and

number of elements, and whether an element is empty or can include text. For each

element, it defines the type and default value. Figure 4.12 gives an example of a schema

that defines the data types and structure of the XML definition of the person structure in

Figure 4.10.

The intention is that a single schema definition may be shared by many different

documents. An XML document that is defined to conform to a particular schema may

also be validated by means of that schema. For example, the sender of a SOAP message

may use an XML schema to encode it, and the recipient will use the same XML schema

to validate and decode it.

Figure 4.11 Illustration of the use of a namespace in the Person structure

<person pers:id="123456789" xmlns:pers = "http://www.cdk5.net/person">

<pers:name> Smith </pers:name>

<pers:place> London </pers:place >

<pers:year> 1984 </pers:year>

</person>

Figure 4.12 An XML schema for the Person structure

<xsd:schema xmlns:xsd = URL of XML schema definitions >

<xsd:element name= "person" type ="personType" />

<xsd:complexType name="personType">

<xsd:sequence>

<xsd:element name = "name" type="xs:string"/>

<xsd:element name = "place" type="xs:string"/>

<xsd:element name = "year" type="xs:positiveInteger"/>

</xsd:sequence>

<xsd:attribute name= "id" type = "xs:positiveInteger"/>

</xsd:complexType>

</xsd:schema>184 CHAPTER 4 INTERPROCESS COMMUNICATION

Document type definitions: Document type definitions (DTDs) [www.w3.org VI] were

provided as a part of the XML 1.0 specification for defining the structure of XML

documents and are still widely used for that purpose. The syntax of DTDs is different

from the rest of XML and it is quite limited in what it can specify; for example, it cannot

describe data types and its definitions are global, preventing element names from being

duplicated. DTDs are not used for defining web services, although they may still be used

to define documents that are transmitted by web services.

APIs for accessing XML • XML parsers and generators are available for most

commonly used programming languages. For example, there is Java software for

writing out Java objects as XML (marshalling) and for creating Java objects from such

structures (unmarshalling). Similar software is available in Python for Python data types

and objects.

4.3.4 Remote object references

This section applies only to languages such as Java and CORBA that support the

distributed object model. It is not relevant to XML.

When a client invokes a method in a remote object, an invocation message is sent

to the server process that hosts the remote object. This message needs to specify which

particular object is to have its method invoked. A remote object reference is an identifier

for a remote object that is valid throughout a distributed system. A remote object

reference is passed in the invocation message to specify which object is to be invoked.

Chapter 5 explains that remote object references are also passed as arguments and

returned as results of remote method invocations, that each remote object has a single

remote object reference and that remote object references can be compared to see

whether they refer to the same remote object. Here, we discuss the external

representation of remote object references.

Remote object references must be generated in a manner that ensures uniqueness

over space and time. In general, there may be many processes hosting remote objects,

so remote object references must be unique among all of the processes in the various

computers in a distributed system. Even after the remote object associated with a given

remote object reference is deleted, it is important that the remote object reference is not

reused, because its potential invokers may retain obsolete remote object references. Any

attempt to invoke a deleted object should produce an error rather than allow access to a

different object.

There are several ways to ensure that a remote object reference is unique. One way

is to construct a remote object reference by concatenating the Internet address of its host

computer and the port number of the process that created it with the time of its creation

and a local object number. The local object number is incremented each time an object

is created in that process.

The port number and time together produce a unique process identifier on that

computer. With this approach, remote object references might be represented with a

format such as that shown in Figure 4.13. In the simplest implementations of RMI,

remote objects live only in the process that created them and survive only as long as that

process continues to run. In such cases, the remote object reference can be used as the

address of the remote object. In other words, invocation messages are sent to the InternetSECTION 4.4 MULTICAST COMMUNICATION 185

address in the remote reference and to the process on that computer using the given port

number.

To allow remote objects to be relocated into a different process on a different

computer, the remote object reference should not be used as the address of the remote

object. Section 8.3.3 discusses a form of remote object reference that allows objects to

be activated in different servers throughout its lifetime.

The peer-to-peer overlay systems described in Chapter 10 use a form of remote

object reference that is completely independent of location. Messages are routed to

resources by means of a distributed routing algorithm.

The last field of the remote object reference shown in Figure 4.13 contains some

information about the interface of the remote object, for example, the interface name.

This information is relevant to any process that receives a remote object reference as an

argument or as the result of a remote invocation, because it needs to know about the

methods offered by the remote object. This point is explained again in Section 5.4.2.

4.4 Multicast communication

The pairwise exchange of messages is not the best model for communication from one

process to a group of other processes, which may be necessary, for example, when a

service is implemented as a number of different processes in different computers,

perhaps to provide fault tolerance or to enhance availability. A multicast operation is

more appropriate – this is an operation that sends a single message from one process to

each of the members of a group of processes, usually in such a way that the membership

of the group is transparent to the sender. There is a range of possibilities in the desired

behaviour of a multicast. The simplest multicast protocol provides no guarantees about

message delivery or ordering.

Multicast messages provide a useful infrastructure for constructing distributed

systems with the following characteristics:

1. Fault tolerance based on replicated services: A replicated service consists of a

group of servers. Client requests are multicast to all the members of the group,

each of which performs an identical operation. Even when some of the members

fail, clients can still be served.

2. Discovering services in spontaneous networking: Section 1.3.2 defines service

discovery in the context of spontaneous networking. Multicast messages can be

used by servers and clients to locate available discovery services in order to

register their interfaces or to look up the interfaces of other services in the

distributed system.

Figure 4.13 Representation of a remote object reference

32 bits 32 bits 32 bits 32 bits

Internet address port number time object number interface of

remote object186 CHAPTER 4 INTERPROCESS COMMUNICATION

3. Better performance through replicated data: Data are replicated to increase the

performance of a service – in some cases replicas of the data are placed in users’

computers. Each time the data changes, the new value is multicast to the processes

managing the replicas.

4. Propagation of event notifications: Multicast to a group may be used to notify

processes when something happens. For example, in Facebook, when someone

changes their status, all their friends receive notifications. Similarly, publishsubscribe protocols may make use of group multicast to disseminate events to

subscribers (see Chapter 6).

In this section introduce IP multicast and then review the needs of the above uses of

group communication to see which of them can be satisfied by IP multicast. For those

that cannot, we propose some further properties for group communication protocols in

addition to those provided by IP multicast.

4.4.1 IP multicast – An implementation of multicast communication

This section discusses IP multicast and presents Java’s API to it via the MulticastSocket

class.

IP multicast • IP multicast is built on top of the Internet Protocol (IP). Note that IP

packets are addressed to computers – ports belong to the TCP and UDP levels. IP

multicast allows the sender to transmit a single IP packet to a set of computers that form

a multicast group. The sender is unaware of the identities of the individual recipients and

of the size of the group. A multicast group is specified by a Class D Internet address (see

Figure 3.15) – that is, an address whose first 4 bits are 1110 in IPv4.

Being a member of a multicast group allows a computer to receive IP packets sent

to the group. The membership of multicast groups is dynamic, allowing computers to

join or leave at any time and to join an arbitrary number of groups. It is possible to send

datagrams to a multicast group without being a member.

At the application programming level, IP multicast is available only via UDP. An

application program performs multicasts by sending UDP datagrams with multicast

addresses and ordinary port numbers. It can join a multicast group by making its socket

join the group, enabling it to receive messages to the group. At the IP level, a computer

belongs to a multicast group when one or more of its processes has sockets that belong

to that group. When a multicast message arrives at a computer, copies are forwarded to

all of the local sockets that have joined the specified multicast address and are bound to

the specified port number. The following details are specific to IPv4:

Multicast routers: IP packets can be multicast both on a local network and on the

wider Internet. Local multicasts use the multicast capability of the local network, for

example, of an Ethernet. Internet multicasts make use of multicast routers, which

forward single datagrams to routers on other networks, where they are again

multicast to local members. To limit the distance of propagation of a multicast

datagram, the sender can specify the number of routers it is allowed to pass – called

the time to live, or TTL for short. To understand how routers know which other

routers have members of a multicast group, see Comer [2007].SECTION 4.4 MULTICAST COMMUNICATION 187

Multicast address allocation: As discussed in Chapter 3, Class D addresses (that is,

addresses in the range 224.0.0.0 to 239.255.255.255) are reserved for multicast traffic

and managed globally by the Internet Assigned Numbers Authority (IANA). The

management of this address space is reviewed annually, with current practice

documented in RPC 3171 [Albanna et al. 2001]. This document defines a partitioning

of this address space into a number of blocks, including:

• Local Network Control Block (224.0.0.0 to 224.0.0.225), for multicast traffic

within a given local network.

• Internet Control Block (224.0.1.0 to 224.0.1.225).

• Ad Hoc Control Block (224.0.2.0 to 224.0.255.0), for traffic that does not fit

any other block.

• Administratively Scoped Block (239.0.0.0 to 239.255.255.255), which is used

to implement a scoping mechanism for multicast traffic (to constrain

propagation).

Multicast addresses may be permanent or temporary. Permanent groups exist even

when there are no members – their addresses are assigned by IANA and span the

various blocks mentioned above. For example, 224.0.1.1 in the Internet block is

reserved for the Network Time Protocol (NTP), as discussed in Chapter 14, and the

range 224.0.6.000 to 224.0.6.127 in the ad hoc block is reserved for the ISIS project

(see Chapters 6 and 18). Addresses are reserved for a variety of purposes, from

specific Internet protocols to given organizations that make heavy use of multicast

traffic, including multimedia broadcasters and financial institutions. A full list of

reserved addresses can be seen on the IANA web site [www.iana.org II].

The remainder of the multicast addresses are available for use by temporary

groups, which must be created before use and cease to exist when all the members

have left. When a temporary group is created, it requires a free multicast address to

avoid accidental participation in an existing group. The IP multicast protocol does not

directly address this issue. If used locally, relatively simple solutions are possible –

for example setting the TTL to a small value, making collisions with other groups

unlikely. However, programs using IP multicast throughout the Internet require a

more sophisticated solution to this problem. RFC 2908 [Thaler et al. 2000] describes

a multicast address allocation architecture (MALLOC) for Internet-wide

applications, that allocates unique addresses for a given period of time and in a given

scope. As such, the proposal is intrinsically bound with the scoping mechanisms

mentioned above. A client-server solution is adopted whereby clients request a

multicast address from a multicast address allocation server (MAAS), which must

then communicate across domains to ensure allocations are unique for the given

lifetime and scope.

Failure model for multicast datagrams • Datagrams multicast over IP multicast have the

same failure characteristics as UDP datagrams – that is, they suffer from omission

failures. The effect on a multicast is that messages are not guaranteed to be delivered to

any particular group member in the face of even a single omission failure. That is, some

but not all of the members of the group may receive it. This can be called unreliable

multicast, because it does not guarantee that a message will be delivered to any member

of a group. Reliable multicast is discussed in Chapter 15.188 CHAPTER 4 INTERPROCESS COMMUNICATION

Java API to IP multicast • The Java API provides a datagram interface to IP multicast

through the class MulticastSocket, which is a subclass of DatagramSocket with the

additional capability of being able to join multicast groups. The class MulticastSocket

provides two alternative constructors, allowing sockets to be created to use either a

specified local port (6789, in Figure 4.14) or any free local port. A process can join a

multicast group with a given multicast address by invoking the joinGroup method of its

multicast socket. Effectively, the socket joins a multicast group at a given port and it will

receive datagrams sent by processes on other computers to that group at that port. A

process can leave a specified group by invoking the leaveGroup method of its multicast

socket.

In the example in Figure 4.14, the arguments to the main method specify a

message to be multicast and the multicast address of a group (for example, "228.5.6.7").

After joining that multicast group, the process makes an instance of DatagramPacket

containing the message and sends it through its multicast socket to the multicast group

address at port 6789. After that, it attempts to receive three multicast messages from its

Figure 4.14 Multicast peer joins a group and sends and receives datagrams

import java.net.\*;

import java.io.\*;

public class MulticastPeer{

public static void main(String args[]){

// args give message contents & destination multicast group (e.g. "228.5.6.7")

MulticastSocket s =null;

try {

InetAddress group = InetAddress.getByName(args[1]);

s = new MulticastSocket(6789);

s.joinGroup(group);

byte [] m = args[0].getBytes();

DatagramPacket messageOut =

new DatagramPacket(m, m.length, group, 6789);

s.send(messageOut);

byte[] buffer = new byte[1000];

for(int i=0; i< 3; i++) { // get messages from others in group

DatagramPacket messageIn =

new DatagramPacket(buffer, buffer.length);

s.receive(messageIn);

System.out.println("Received:" + new String(messageIn.getData()));

}

s.leaveGroup(group);

} catch (SocketException e){System.out.println("Socket: " + e.getMessage());

} catch (IOException e){System.out.println("IO: " + e.getMessage());

} finally { if(s != null) s.close();}

}

}SECTION 4.4 MULTICAST COMMUNICATION 189

peers via its socket, which also belongs to the group on the same port. When several

instances of this program are run simultaneously on different computers, all of them join

the same group, and each of them should receive its own message and the messages from

those that joined after it.

The Java API allows the TTL to be set for a multicast socket by means of the

setTimeToLive method. The default is 1, allowing the multicast to propagate only on the

local network.

An application implemented over IP multicast may use more than one port. For

example, the MultiTalk [mbone] application, which allows groups of users to hold textbased conversations, has one port for sending and receiving data and another for

exchanging control data.

4.4.2 Reliability and ordering of multicast

The previous section stated the failure model for IP multicast, which suffers from

omission failures. A datagram sent from one multicast router to another may be lost, thus

preventing all recipients beyond that router from receiving the message. Also, when a

multicast on a local area network uses the multicasting capabilities of the network to

allow a single datagram to arrive at multiple recipients, any one of those recipients may

drop the message because its buffer is full.

Another factor is that any process may fail. If a multicast router fails, the group

members beyond that router will not receive the multicast message, although local

members may do so.

Ordering is another issue. IP packets sent over an internetwork do not necessarily

arrive in the order in which they were sent, with the possible effect that some group

members receive datagrams from a single sender in a different order from other group

members. In addition, messages sent by two different processes will not necessarily

arrive in the same order at all the members of the group.

Some examples of the effects of reliability and ordering • We now consider the effect of

the failure semantics of IP multicast on the four examples of the use of replication in the

introduction to Section 4.4.

1. Fault tolerance based on replicated services: Consider a replicated service that

consists of the members of a group of servers that start in the same initial state and

always perform the same operations in the same order, so as to remain consistent

with one another. This application of multicast requires that either all of the

replicas or none of them should receive each request to perform an operation – if

one of them misses a request, it will become inconsistent with the others. In most

cases, this service would require that all members receive request messages in the

same order as one another.

3. Discovering services in spontaneous networking: One way for a process to

discover services in spontaneous networking is to multicast requests at periodic

intervals, and for the available services to listen for those multicasts and respond.

An occasional lost request is not an issue when discovering services. In fact, Jini

uses IP multicast in its protocol for discovering services. This is described in

Section 19.2.1.190 CHAPTER 4 INTERPROCESS COMMUNICATION

3. Better performance through replicated data: Consider the case where the

replicated data itself, rather than operations on the data, are distributed by means

of multicast messages. The effect of lost messages and inconsistent ordering

would depend on the method of replication and the importance of all replicas

being totally up-to-date.

4. Propagation of event notifications: The particular application determines the

qualities required of multicast. For example, the Jini lookup services use IP

multicast to announce their existence (see Section 19.2.1).

These examples suggest that some applications require a multicast protocol that is more

reliable than IP multicast. In particular, there is a need for reliable multicast, in which

any message transmitted is either received by all members of a group or by none of them.

The examples also suggest that some applications have strong requirements for

ordering, the strictest of which is called totally ordered multicast, in which all of the

messages transmitted to a group reach all of the members in the same order.

Chapter 15 will define and show how to implement reliable multicast and various

useful ordering guarantees, including totally ordered multicast.

4.5 Network virtualization: Overlay networks

The strength of the Internet communication protocols is that they provide, through their

API (Section 4.2), a very effective set of building blocks for the construction of

distributed software. However, a growing range of different classes of application

(including, for example, peer-to-peer file sharing and Skype) coexist in the Internet. It

would be impractical to attempt to alter the Internet protocols to suit each of the many

applications running over them – what might enhance one of them could be detrimental

to another. In addition, the IP transport service is implemented over a large and everincreasing number of network technologies. These two factors have led to the interest in

network virtualization.

Network virtualization [Petersen et al. 2005] is concerned with the construction of

many different virtual networks over an existing network such as the Internet. Each

virtual network can be designed to support a particular distributed application. For

example, one virtual network might support multimedia streaming, as in BBC iPlayer,

BoxeeTV [boxee.tv] or Hulu [hulu.com], and coexist with another that supports a

multiplayer online game, both running over the same underlying network. This suggests

an answer to the dilemma raised by Salzer’s end-to-end argument (see Section 2.3.3): an

application-specific virtual network can be built above an existing network and

optimized for that particular application, without changing the characteristics of the

underlying network.

Chapter 3 showed that computer networks have addressing schemes, protocols

and routing algorithms; similarly, each virtual network has its own particular addressing

scheme, protocols and routing algorithms, but redefined to meet the needs of particular

application classes.SECTION 4.5 NETWORK VIRTUALIZATION: OVERLAY NETWORKS 191

4.5.1 Overlay networks

An overlay network is a virtual network consisting of nodes and virtual links, which sits

on top of an underlying network (such as an IP network) and offers something that is not

otherwise provided:

• a service that is tailored towards the needs of a class of application or a particular

higher-level service – for example, multimedia content distribution;

• more efficient operation in a given networked environment – for example routing

in an ad hoc network;

• an additional feature – for example, multicast or secure communication.

This leads to a wide variety of types of overlay as captured by Figure 4.15. Overlay

networks have the following advantages:

• They enable new network services to be defined without requiring changes to the

underlying network, a crucial point given the level of standardization in this area

and the difficulties of amending underlying router functionality.

• They encourage experimentation with network services and the customization of

services to particular classes of application.

• Multiple overlays can be defined and can coexist, with the end result being a more

open and extensible network architecture.

The disadvantages are that overlays introduce an extra level of indirection (and hence

may incur a performance penalty) and they add to the complexity of network services

when compared, for example, to the relatively simple architecture of TCP/IP networks.

Overlays can be related to the familiar concept of layers (as introduced in Chapters

2 and 3). Overlays are layers, but layers that exist outside the standard architecture (such

as the TCP/IP stack) and exploit the resultant degrees of freedom. In particular, overlay

developers are free to redefine the core elements of a network as mentioned above,

including the mode of addressing, the protocols employed and the approach to routing,

often introducing radically different approaches more tailored towards particular

application classes of operating environments. For example, distributed hash tables

introduce a style of addressing based on a keyspace and also build a topology in such a

way that a node in the topology either owns the key or has a link to a node that is closer

to the owner (a style of routing known as key-based routing). The topology is most

commonly in the form of a ring.

We exemplify the successful use of an overlay network by discussing Skype.

Further examples of overlays will be given throughout the book. For example, Chapter

10 presents details of the protocols and structures adopted by peer-to-peer file sharing,

along with further information on distributed hash tables. Chapter 19 considers both

wireless ad hoc networks and disruption-tolerant networks in the context of mobile and

ubiquitous computing and Chapter 20 examines overlay support for multimedia

streaming.192 CHAPTER 4 INTERPROCESS COMMUNICATION

Figure 4.15 Types of overlay

Motivation Type Description

Tailored for

application needs

Distributed hash tables One of the most prominent classes of overlay

network, offering a service that manages a

mapping from keys to values across a potentially

large number of nodes in a completely

decentralized manner (similar to a standard hash

table but in a networked environment).

Peer-to-peer file

sharing

Overlay structures that focus on constructing

tailored addressing and routing mechanisms to

support the cooperative discovery and use (for

example, download) of files.

Content distribution

networks

Overlays that subsume a range of replication,

caching and placement strategies to provide

improved performance in terms of content

delivery to web users; used for web acceleration

and to offer the required real-time performance

for video streaming [www.kontiki.com].

Tailored for

network style

Wireless ad hoc

networks

Network overlays that provide customized

routing protocols for wireless ad hoc networks,

including proactive schemes that effectively

construct a routing topology on top of the

underlying nodes and reactive schemes that

establish routes on demand typically supported

by flooding.

Disruption-tolerant

networks

Overlays designed to operate in hostile

environments that suffer significant node or link

failure and potentially high delays.

Offering additional

features

Multicast One of the earliest uses of overlay networks in

the Internet, providing access to multicast services where multicast routers are not available;

builds on the work by Van Jacobsen, Deering

and Casner with their implementation of the

MBone (or Multicast Backbone) [mbone].

Resilience Overlay networks that seek an order of

magnitude improvement in robustness and

availability of Internet paths

[nms.csail.mit.edu].

Security Overlay networks that offer enhanced security

over the underling IP network, including virtual

private networks, for example, as discussed in

Section 3.4.8.SECTION 4.5 NETWORK VIRTUALIZATION: OVERLAY NETWORKS 193

4.5.2 Skype: An example of an overlay network

Skype is a peer-to-peer application offering Voice over IP (VoIP). It also includes

instant messaging, video conferencing and interfaces to the standard telephony service

through SkypeIn and SkypeOut. The software was developed by Kazaa in 2003 and

hence shares many of the characteristics of the Kazaa peer-to-peer file-sharing

application [Leibowitz et al. 2003]. It is widely deployed, with an estimated 370 million

users as of the start of 2009.

Skype is an excellent case study of the use of overlay networks in real-world (and

large-scale) systems, indicating how advanced functionality can be provided in an

application-specific manner and without modification of the core architecture of the

Internet. Skype is a virtual network in that it establishes connections between people

(Skype subscribers who are currently active). No IP address or port is required to

establish a call. The architecture of the virtual network supporting Skype is not widely

publicized but researchers have studied Skype through a variety of methods, including

traffic analysis, and its principles are now in the public domain. Much of the detail of

the description that follows is taken from the paper by Baset and Schulzrinne [2006],

which contains a detailed study of the behaviour of Skype.

Figure 4.16 Skype overlay architecture

SN

Super node

Ordinary host

Skype

login server

SN

SN

SN

SN194 CHAPTER 4 INTERPROCESS COMMUNICATION

Skype architecture • Skype is based on a peer-to-peer infrastructure consisting of

ordinary users’ machines (referred to as hosts) and super nodes – super nodes are

ordinary Skype hosts that happen to have sufficient capabilities to carry out their

enhanced role. Super nodes are selected on demand based a range of criteria including

bandwidth available, reachability (the machine must have a global IP address and not be

hidden behind a NAT-enabled router, for example) and availability (based on the length

of time that Skype has been running continuously on that node). This overall structure

is captured in Figure 4.16.

User connection • Skype users are authenticated via a well-known login server. They

then make contact with a selected super node. To achieve this, each client maintains a

cache of super node identities (that is, IP address and port number pairs). At first login

this cache is filled with the addresses of around seven super nodes, and over time the

client builds and maintains a much larger set (perhaps several hundred).

Search for users • The main goal of super nodes is to perform the efficient search of the

global index of users, which is distributed across the super nodes. The search is

orchestrated by the client’s chosen super node and involves an expanding search of other

super nodes until the specified user is found. On average, eight super nodes are

contacted. A user search typically takes between three and four seconds to complete for

hosts that have a global IP address (and slightly longer, five to six seconds, if behind a

NAT-enabled router). From experiments, it appears that intermediary nodes involved in

the search cache the results to improve performance.

Voice connection • Once the required user is discovered, Skype establishes a voice

connection between the two parties using TCP for signalling call requests and

terminations and either UDP or TCP for the streaming audio. UDP is preferred but TCP,

along with the use of an intermediary node, is used in certain circumstances to

circumvent firewalls (see Baset and Schulzrinne [2006] for details). The software used

for encoding and decoding audio plays a key part in providing the excellent call quality

normally attained using Skype, and the associated algorithms are carefully tailored to

operate in Internet environments at 32 kbps and above.

4.6 Case study: MPI

Message passing was introduced in Section 4.2.1, which outlines the basic principles of

exchanging messages between two processes using send and receive operations. The

synchronous variant of message passing is realised by blocking send and receive calls,

whereas the asynchronous variant requires a non-blocking form of send. The end result

is a paradigm for distributed programming that is lightweight, efficient and in many

ways minimal.

This style of distributed programming is attractive in classes of system where

performance is paramount, most notably in high-performance computing. In this

section, we present a case study of the Message Passing Interface standard, developed

by the high performance computing community. MPI was first introduced in 1994 by

the MPI Forum [www.mpi-forum.org] as a reaction against the wide variety of

proprietary approaches that were in use for message passing in this field. The standardSECTION 4.6 CASE STUDY: MPI 195

has also been strongly influential in Grid computing (discussed in Chapter 9), for

example through the development of GridMPI [www.gridmpi.org]. The goal of the MPI

Forum was to retain the inherent simplicity, practicality and efficiency of the messagepassing approach but enhance this with portability through presenting a standardized

interface independent of the operating system or programming language-specific socket

interface. MPI was also designed to be flexible, and the result is a comprehensive

specification of message passing in all its variants (with over 115 operations).

Applications use the MPI interface via a message-passing library available for a variety

of operating systems and programming languages, including C++ and Fortran.

The underlying architectural model for MPI is relatively simple and captured in

Figure 4.17. This is similar to the model introduced in Section 4.2.1, but with the added

dimension of explicitly having MPI library buffers in both the sender and the receiver,

managed by the MPI library and used to hold data in transit. Note that this figure shows

one pathway from the sender to the receiver via the receiver’s MPI library buffer (other

options, for example using the sender’s MPI library buffer, will become apparent

below).

To provide a flavour of this complexity, let us examine a number of the variants

of send summarized in Figure 4.18. This is a refinement of the view of message passing

as presented in Section 4.2.1, offering more choice and control and effectively

separating the semantics of synchronous/asynchronous and blocking/non-blocking

message passing.

We start by examining the four blocking operations presented in the associated

column of Figure 4.18. The key to understanding this set of operations is to appreciate

that blocking is interpreted as ‘blocked until it is safe to return’, in the sense that

application data has been copied into the MPI system and hence is in transit or delivered

and therefore the application buffer can be reused (for example, for the next send

operation). This then enables various interpretations of ‘being safe to return’. The

MPI\_Send operation is a generic operation that simply requires that this level of safety

is provided (in practice, this is often implemented using MPI\_Ssend). MPI\_Ssend is

exactly the same as synchronous (and blocking) message passing as introduced in

Section 4.2.1, with safety interpreted as delivered, whereas MPI\_Bsend has weaker

Figure 4.17 An overview of point-to-point communication in MPI

Message

Process p Process q

MPI library buffer

send m receive196 CHAPTER 4 INTERPROCESS COMMUNICATION

semantics in that the message is considered safe when it has been has been copied into

the preallocated MPI library buffer and is still in transit. MPI\_Rsend is a rather curious

operation in which the programmer specifies that they know that the receiver is ready to

receive the message. If this is known, the underlying implementation can be optimized

in that there is no need to check if there is a buffer available to receive the message,

avoiding a handshake. This is clearly a rather dangerous operation that will fail if the

assumption about being ready is invalid. From the figure, it is possible to observe the

elegant symmetry for non-blocking send operations, this time defined over the

semantics of the associated MPI\_Wait and MPI\_Test operations (note also the consistent

naming convention across all the operations).

The standard also supports both blocking and non-blocking receive (MPI\_recv

and MPI\_Irecv, respectively), and the variants of send and receive can be paired in any

combination, offering the programmer rich control over the semantics of message

passing. In addition, the standard defines rich patterns of multiway communication

(referred to as collective communication) including, for example, scatter (one to many)

and gather (many to one) operations.

Figure 4.18 Selected send operations in MPI

Send operations Blocking Non-blocking

Generic MPI\_Send: the sender blocks until

it is safe to return – that is, until the

message is in transit or delivered

and the sender’s application buffer

can therefore be reused.

MPI\_Isend: the call returns

immediately and the programmer is

given a communication request

handle, which can then be used to

check the progress of the call via

MPI\_Wait or MPI\_Test.

Synchronous MPI\_Ssend: the sender and receiver

synchronize and the call only

returns when the message has been

delivered at the receiving end.

MPI\_Issend: as with MPI\_Isend,

but with MPI\_Wait and MPI\_Test

indicating whether the message has

been delivered at the receive end.

Buffered MPI\_Bsend: the sender explicitly

allocates an MPI buffer library

(using a separate

MPI\_Buffer\_attach call) and the

call returns when the data is

successfully copied into this buffer.

MPI\_Ibsend: as with MPI\_Isend

but with MPI\_Wait and MPI\_Test

indicating whether the message has

been copied into the sender’s MPI

buffer and hence is in transit.

Ready MPI\_Rsend: the call returns when

the sender’s application buffer can

be reused (as with MPI\_Send), but

the programmer is also indicating to

the library that the receiver is ready

to receive the message, resulting in

potential optimization of the

underlying implementation.

MPI\_Irsend: the effect is as with

MPI\_Isend, but as with

MPI\_Rsend, the programmer is

indicating to the underlying

implementation that the receiver is

guaranteed to be ready to receive

(resulting in the same

optimizations),SECTION 4.7 SUMMARY 197

4.7 Summary

The first section of this chapter showed that the Internet transmission protocols provide

two alternative building blocks from which application protocols may be constructed.

There is an interesting trade-off between the two protocols: UDP provides a simple

message-passing facility that suffers from omission failures but carries no built-in

performance penalties, on the other hand, in good conditions TCP guarantees message

delivery, but at the expense of additional messages and higher latency and storage costs.

The second section showed three alternative styles of marshalling. CORBA and

its predecessors choose to marshal data for use by recipients that have prior knowledge

of the types of its components. In contrast, when Java serializes data, it includes full

information about the types of its contents, allowing the recipient to reconstruct it purely

from the content. XML, like Java, includes full type information. Another big difference

is that CORBA requires a specification of the types of data items to be marshalled (in

IDL) in order to generate the marshalling and unmarshalling methods, whereas Java

uses reflection in order to serialize objects and deserialize their serial form. But a variety

of different means are used for generating XML, depending on the context. For example,

many programming languages, including Java, provide processors for translating

between XML and language-level objects.

Multicast messages are used in communication between the members of a group

of processes. IP multicast provides a multicast service for both local area networks and

the Internet. This form of multicast has the same failure semantics as UDP datagrams,

but in spite of suffering from omission failures it is a useful tool for many applications

of multicast. Some other applications have stronger requirements – in particular, that

multicast delivery should be atomic; that is, it should have all-or-nothing delivery.

Further requirements on multicast are related to the ordering of messages, the strongest

of which requires that all members of a group receive all of the messages in the same

order.

Multicast can also be supported by overlay networks in cases where, for example,

IP multicast is not supported. More generally, overlay networks offer a service of

virtualization of the network architecture, allowing specialist network services to be

created on top of underlying networking infrastructure, (for example, UDP or TCP).

Overlay networks partially address the problems associated with Saltzer’s end-to-end

argument by allowing the generation of more application-specific network abstractions.

The chapter concluded with a case study of the MPI specification, developed by

the high-performance computing community and featuring flexible support for message

passing together with additional support for multiway message passing.198 CHAPTER 4 INTERPROCESS COMMUNICATION

EXERCISES

4.1 In synchronous communication, how do the send and receive operations work? page 164

4.2 What is socket abstraction? Name the main protocols used in interprocess

communication. page 165

4.3 The programs in Figure 4.3 and Figure 4.4 are available at www.cdk5.net/ipc. Use them

to make a test kit to determine the conditions in which datagrams are sometimes dropped.

Hint: the client program should be able to vary the number of messages sent and their size;

the server should detect when a message from a particular client is missed. page 166

4.4 Use the program in Figure 4.3 to make a client program that repeatedly reads a line of

input from the user, sends it to the server in a UDP datagram message, then receives a

message from the server. The client sets a timeout on its socket so that it can inform the

user when the server does not reply. Test this client program with the server program in

Figure 4.4. page 166

4.5 The programs in Figure 4.5 and Figure 4.6 are available at www.cdk5.net/ipc. Modify

them so that the client repeatedly takes a line of user’s input and writes it to the stream

and the server reads repeatedly from the stream, printing out the result of each read.

Make a comparison between sending data in UDP datagram messages and over a stream.

page 169

4.6 Write one server and one client program. Test whether more than one client program can

simultaneously interact with the server program. page 169

4.7 Describe the three alternative approaches to external data representation and

marshalling. page 175

4.8 Sun XDR aligns each primitive value on a 4-byte boundary, whereas CORBA CDR

aligns a primitive value of size n on an n-byte boundary. Discuss the trade-offs in

choosing the sizes occupied by primitive values. page 176

4.9 What are the different ways in which CORBA CDR can represent constructed types?

page 176

4.10 Write an algorithm in pseudo-code to describe the serialization procedure described in

Section 4.3.2. The algorithm should show when handles are defined or substituted for

classes and instances. Describe the serialized form that your algorithm would produce

when serializing an instance of the following class, Couple:

class Couple implements Serializable{

private Person one;

private Person two;

public Couple(Person a, Person b) {

one = a;

two = b;

}

} page 178EXERCISES 199

4.11 Write an algorithm in pseudo-code to describe deserialization of the serialized form

produced by the algorithm defined in Exercise 4.10. Hint: use reflection to create a class

from its name, to create a constructor from its parameter types and to create a new

instance of an object from the constructor and the argument values. page 178

4.12 Why can’t binary data be represented directly in XML, for example, by representing it

as Unicode byte values? XML elements can carry strings represented as base64. Discuss

the advantages or disadvantages of using this method to represent binary data.

page 180

4.13 Define a class whose instances represent remote object references. It should contain

information similar to that shown in Figure 4.13 and should provide access methods

needed by higher-level protocols (see request-reply in Chapter 5, for example). Explain

how each of the access methods will be used by that protocol. Give a justification for the

type chosen for the instance variable containing information about the interface of the

remote object. page 184

4.14 IP multicast provides a service that suffers from omission failures. Make a test kit,

possibly based on the program in Figure 4.14, to discover the conditions under which a

multicast message is sometimes dropped by one of the members of the multicast group.

The test kit should be designed to allow for multiple sending processes. page 186

4.15 Outline the design of a scheme that uses message retransmissions with IP multicast to

overcome the problem of dropped messages. Your scheme should take the following

points into account:

i) There may be multiple senders.

ii) Generally only a small proportion of messages are dropped.

iii) Recipients may not necessarily send a message within any particular time limit.

Assume that messages that are not dropped arrive in sender order. page 189

4.16 What are the functions and responsibilities of a multicast address allocation architecture

(MALLOC) and a multicast address allocation server (MAAS)? pages 187, 188

4.17 Devise a scenario in which multicasts sent by different clients are delivered to two group

members in different orders. Assume that some form of message retransmission is in

use, but that messages that are not dropped arrive in sender order. Suggest how

recipients might remedy this situation. page 189

4.18 Revisit the Internet architecture as introduced in Chapter 3 (see Figures 3.12 and 3.14).

What impact does the introduction of overlay networks have on this architecture, and in

particular on the programmer’s conceptual view of the Internet? page 191

4.19 What are the main characteristics of Skype? page 193

4.20 As discussed in Section 4.6, MPI offers a number of variants of send including the

MPI\_Rsend operation, which assumes the receiver is ready to receive at the time of

sending. What optimizations in implementation are possible if this assumption is correct

and what are the repercussions of this assumption being false? page 196This page intentionally left blank201

5

REMOTE INVOCATION

5.1 Introduction

5.2 Request-reply protocols

5.3 Remote procedure call

5.4 Remote method invocation

5.5 Case study: Java RMI

5.6 Summary

This chapter steps through the remote invocation paradigms introduced in Chapter 2

(indirect communication techniques are addressed in Chapter 6). The chapter starts by

examining the most primitive service, request-reply communication, which represents

relatively minor enhancements to the underlying interprocess communication primitives

discussed in Chapter 4. The chapter then continues by examining the two most prominent

remote invocation techniques for communication in distributed systems:

• The remote procedure call (RPC) approach extends the common programming

abstraction of the procedure call to distributed environments, allowing a calling

process to call a procedure in a remote node as if it is local.

• Remote method invocation (RMI) is similar to RPC but for distributed objects, with

added benefits in terms of using object-oriented programming concepts in

distributed systems and also extending the concept of an object reference to the

global distributed environments, and allowing the use of object references as

parameters in remote invocations.

The chapter also features Java RMI as a case study of the remote method invocation

approach (further insight can also be gained in Chapter 8, where we look at CORBA).202 CHAPTER 5 REMOTE INVOCATION

5.1 Introduction

This chapter is concerned with how processes (or entities at a higher level of abstraction

such as objects or services) communicate in a distributed system, examining, in

particular, the remote invocation paradigms defined in Chapter 2:

• Request-reply protocols represent a pattern on top of message passing and support

the two-way exchange of messages as encountered in client-server computing. In

particular, such protocols provide relatively low-level support for requesting the

execution of a remote operation, and also provide direct support for RPC and

RMI, discussed below.

• The earliest and perhaps the best-known example of a more programmer-friendly

model was the extension of the conventional procedure call model to distributed

systems (the remote procedure call, or RPC, model), which allows client

programs to call procedures transparently in server programs running in separate

processes and generally in different computers from the client.

• In the 1990s, the object-based programming model was extended to allow objects

in different processes to communicate with one another by means of remote

method invocation (RMI). RMI is an extension of local method invocation that

allows an object living in one process to invoke the methods of an object living in

another process.

Note that we use the term ‘RMI’ to refer to remote method invocation in a generic way

– this should not be confused with particular examples of remote method invocation

such as Java RMI.

Returning to the diagram first introduced in Chapter 4 (and reproduced in Figure

5.1), this chapter, together with Chapter 6, continues our study of middleware concepts

by focusing on the layer above interprocess communication. In particular, Sections 5.2

through 5.4 focus on the styles of communication listed above, with Section 5.5

providing a more complex case study, Java RMI.

Figure 5.1 Middleware layers

Applications, services

Middleware

layers

Underlying interprocess communication primitives:

UDP and TCP

This

chapter Remote invocation, indirect communication

Sockets, message passing, multicast support, overlay networks

(and Chapter 6)SECTION 5.2 REQUEST-REPLY PROTOCOLS 203

5.2 Request-reply protocols

This form of communication is designed to support the roles and message exchanges in

typical client-server interactions. In the normal case, request-reply communication is

synchronous because the client process blocks until the reply arrives from the server. It

can also be reliable because the reply from the server is effectively an acknowledgement

to the client. Asynchronous request-reply communication is an alternative that may be

useful in situations where clients can afford to retrieve replies later – see Section 7.5.2.

The client-server exchanges are described in the following paragraphs in terms of

the send and receive operations in the Java API for UDP datagrams, although many

current implementations use TCP streams. A protocol built over datagrams avoids

unnecessary overheads associated with the TCP stream protocol. In particular:

• Acknowledgements are redundant, since requests are followed by replies.

• Establishing a connection involves two extra pairs of messages in addition to the

pair required for a request and a reply.

• Flow control is redundant for the majority of invocations, which pass only small

arguments and results.

The request-reply protocol • The protocol we describe here is based on a trio of

communication primitives, doOperation, getRequest and sendReply, as shown in Figure

5.2. This request-reply protocol matches requests to replies. It may be designed to

provide certain delivery guarantees. If UDP datagrams are used, the delivery guarantees

must be provided by the request-reply protocol, which may use the server reply message

as an acknowledgement of the client request message. Figure 5.3 outlines the three

communication primitives.

The doOperation method is used by clients to invoke remote operations. Its

arguments specify the remote server and which operation to invoke, together with

additional information (arguments) required by the operation. Its result is a byte array

containing the reply. It is assumed that the client calling doOperation marshals the

Figure 5.2 Request-reply communication

Request

Client Server

Reply

message

message getRequest

select operation

sendReply

doOperation

(wait)

(continuation)

•• ••

execute operation204 CHAPTER 5 REMOTE INVOCATION

arguments into an array of bytes and unmarshals the results from the array of bytes that

is returned. The first argument of doOperation is an instance of the class RemoteRef,

which represents references for remote servers. This class provides methods for getting

the Internet address and port of the associated server. The doOperation method sends a

request message to the server whose Internet address and port are specified in the remote

reference given as an argument. After sending the request message, doOperation

invokes receive to get a reply message, from which it extracts the result and returns it to

the caller. The caller of doOperation is blocked until the server performs the requested

operation and transmits a reply message to the client process.

getRequest is used by a server process to acquire service requests, as shown in

Figure 5.3. When the server has invoked the specified operation, it then uses sendReply

to send the reply message to the client. When the reply message is received by the client

the original doOperation is unblocked and execution of the client program continues.

The information to be transmitted in a request message or a reply message is

shown in Figure 5.4. The first field indicates whether the message is a Request or a Reply

message. The second field, requestId, contains a message identifier. A doOperation in

the client generates a requestId for each request message, and the server copies these IDs

into the corresponding reply messages. This enables doOperation to check that a reply

message is the result of the current request, not a delayed earlier call. The third field is

a remote reference. The fourth field is an identifier for the operation to be invoked. For

example, the operations in an interface might be numbered 1, 2, 3, ... , if the client and

server use a common language that supports reflection, a representation of the operation

itself may be put in this field.

Figure 5.3 Operations of the request-reply protocol

public byte[] doOperation (RemoteRef s, int operationId, byte[] arguments)

Sends a request message to the remote server and returns the reply.

The arguments specify the remote server, the operation to be invoked and the

arguments of that operation.

public byte[] getRequest ();

Acquires a client request via the server port.

public void sendReply (byte[] reply, InetAddress clientHost, int clientPort);

Sends the reply message reply to the client at its Internet address and port.

Figure 5.4 Request-reply message structure

messageType int (0=Request, 1= Reply)

requestId int

remoteReference RemoteRef

operationId int or Operation

arguments // array of bytesSECTION 5.2 REQUEST-REPLY PROTOCOLS 205

Message identifiers • Any scheme that involves the management of messages to

provide additional properties such as reliable message delivery or request-reply

communication requires that each message have a unique message identifier by which

it may be referenced. A message identifier consists of two parts:

1. a requestId, which is taken from an increasing sequence of integers by the sending

process;

2. an identifier for the sender process, for example, its port and Internet address.

The first part makes the identifier unique to the sender, and the second part makes it

unique in the distributed system. (The second part can be obtained independently – for

example, if UDP is in use, from the message received.)

When the value of the requestId reaches the maximum value for an unsigned

integer (for example, 232 – 1) it is reset to zero. The only restriction here is that the

lifetime of a message identifier should be much less than the time taken to exhaust the

values in the sequence of integers.

Failure model of the request-reply protocol • If the three primitives doOperation,

getRequest and sendReply are implemented over UDP datagrams, then they suffer from

the same communication failures. That is:

• They suffer from omission failures.

• Messages are not guaranteed to be delivered in sender order.

In addition, the protocol can suffer from the failure of processes (see Section 2.4.2). We

assume that processes have crash failures. That is, when they halt, they remain halted –

they do not produce Byzantine behaviour.

To allow for occasions when a server has failed or a request or reply message is

dropped, doOperation uses a timeout when it is waiting to get the server’s reply

message. The action taken when a timeout occurs depends upon the delivery guarantees

being offered.

Timeouts • There are various options as to what doOperation can do after a timeout.

The simplest option is to return immediately from doOperation with an indication to the

client that the doOperation has failed. This is not the usual approach – the timeout may

have been due to the request or reply message getting lost and in the latter case, the

operation will have been performed. To compensate for the possibility of lost messages,

doOperation sends the request message repeatedly until either it gets a reply or it is

reasonably sure that the delay is due to lack of response from the server rather than to

lost messages. Eventually, when doOperation returns, it will indicate to the client by an

exception that no result was received.

Discarding duplicate request messages • In cases when the request message is

retransmitted, the server may receive it more than once. For example, the server may

receive the first request message but take longer than the client’s timeout to execute the

command and return the reply. This can lead to the server executing an operation more

than once for the same request. To avoid this, the protocol is designed to recognize

successive messages (from the same client) with the same request identifier and to filter

out duplicates. If the server has not yet sent the reply, it need take no special action – it

will transmit the reply when it has finished executing the operation.206 CHAPTER 5 REMOTE INVOCATION

Lost reply messages • If the server has already sent the reply when it receives a

duplicate request it will need to execute the operation again to obtain the result, unless

it has stored the result of the original execution. Some servers can execute their

operations more than once and obtain the same results each time. An idempotent

operation is an operation that can be performed repeatedly with the same effect as if it

had been performed exactly once. For example, an operation to add an element to a set

is an idempotent operation because it will always have the same effect on the set each

time it is performed, whereas an operation to append an item to a sequence is not an

idempotent operation because it extends the sequence each time it is performed. A server

whose operations are all idempotent need not take special measures to avoid executing

its operations more than once.

History • For servers that require retransmission of replies without re-execution of

operations, a history may be used. The term ‘history’ is used to refer to a structure that

contains a record of (reply) messages that have been transmitted. An entry in a history

contains a request identifier, a message and an identifier of the client to which it was

sent. Its purpose is to allow the server to retransmit reply messages when client

processes request them. A problem associated with the use of a history is its memory

cost. A history will become very large unless the server can tell when the messages will

no longer be needed for retransmission.

As clients can make only one request at a time, the server can interpret each

request as an acknowledgement of its previous reply. Therefore the history need contain

only the last reply message sent to each client. However, the volume of reply messages

in a server’s history may still be a problem when it has a large number of clients. This

is compounded by the fact that, when a client process terminates, it does not

acknowledge the last reply it has received – messages in the history are therefore

normally discarded after a limited period of time.

Styles of exchange protocols • Three protocols, that produce differing behaviours in the

presence of communication failures are used for implementing various types of request

behaviour. They were originally identified by Spector [1982]:

• the request (R) protocol;

• the request-reply (RR) protocol;

• the request-reply-acknowledge reply (RRA) protocol.

The messages passed in these protocols are summarized in Figure 5.5. In the R protocol,

a single Request message is sent by the client to the server. The R protocol may be used

when there is no value to be returned from the remote operation and the client requires

no confirmation that the operation has been executed. The client may proceed

immediately after the request message is sent as there is no need to wait for a reply

message. This protocol is implemented over UDP datagrams and therefore suffers from

the same communication failures.

The RR protocol is useful for most client-server exchanges because it is based on

the request-reply protocol. Special acknowledgement messages are not required,

because a server’s reply message is regarded as an acknowledgement of the client’s

request message. Similarly, a subsequent call from a client may be regarded as an

acknowledgement of a server’s reply message. As we have seen, communicationSECTION 5.2 REQUEST-REPLY PROTOCOLS 207

failures due to UDP datagrams being lost may be masked by the retransmission of

requests with duplicate filtering and the saving of replies in a history for retransmission.

The RRA protocol is based on the exchange of three messages: request-replyacknowledge reply. The Acknowledge reply message contains the requestId from the

reply message being acknowledged. This will enable the server to discard entries from

its history. The arrival of a requestId in an acknowledgement message will be

interpreted as acknowledging the receipt of all reply messages with lower requestIds, so

the loss of an acknowledgement message is harmless. Although the exchange involves

an additional message, it need not block the client, as the acknowledgement may be

transmitted after the reply has been given to the client. However it does use processing

and network resources. Exercise 5.10 suggests an optimization to the RRA protocol.

Use of TCP streams to implement the request-reply protocol • Section 4.2.3 mentioned

that it is often difficult to decide on an appropriate size for the buffer in which to receive

datagrams. In the request-reply protocol, this applies to the buffers used by the server to

receive request messages and by the client to receive replies. The limited length of

datagrams (usually 8 kilobytes) may not be regarded as adequate for use in transparent

RMI or RPC systems, since the arguments or results of procedures may be of any size.

The desire to avoid implementing multipacket protocols is one of the reasons for

choosing to implement request-reply protocols over TCP streams, allowing arguments

and results of any size to be transmitted. In particular, Java object serialization is a

stream protocol that allows arguments and results to be sent over streams between the

client and server, making it possible for collections of objects of any size to be

transmitted reliably. If the TCP protocol is used, it ensures that request and reply

messages are delivered reliably, so there is no need for the request-reply protocol to deal

with retransmission of messages and filtering of duplicates or with histories. In addition

the flow-control mechanism allows large arguments and results to be passed without

taking special measures to avoid overwhelming the recipient. Thus the TCP protocol is

chosen for request-reply protocols because it can simplify their implementation. If

successive requests and replies between the same client-server pair are sent over the

same stream, the connection overhead need not apply to every remote invocation. Also,

the overhead due to TCP acknowledgement messages is reduced when a reply message

follows soon after a request message.

Howeever, if the application does not require all of the facilities offered by TCP,

a more efficient, specially tailored protocol can be implemented over UDP. For

example, Sun NFS does not require support for messages of unlimited size, since it

Figure 5.5 RPC exchange protocols

Name Messages sent by

Client Server Client

R Request

RR Request Reply

RRA Request Reply Acknowledge reply208 CHAPTER 5 REMOTE INVOCATION

transmits fixed-size file blocks between client and server. In addition to that, its

operations are designed to be idempotent, so it does not matter if operations are executed

more than once in order to retransmit lost reply messages, making it unnecessary to

maintain a history.

HTTP: An example of a request-reply protocol • Chapter 1 introduced the HyperText

Transfer Protocol (HTTP) used by web browser clients to make requests to web servers

and to receive replies from them. To recap, web servers manage resources implemented

in different ways:

• as data – for example the text of an HTML page, an image or the class of an applet;

• as a program – for example, servlets [java.sun.com III], or PHP or Python

programs that run on the web server.

Client requests specify a URL that includes the DNS hostname of a web server and an

optional port number on the web server as well as the identifier of a resource on that

server.

HTTP is a protocol that specifies the messages involved in a request-reply

exchange, the methods, arguments and results, and the rules for representing

(marshalling) them in the messages. It supports a fixed set of methods (GET, PUT,

POST, etc) that are applicable to all of the server’s resources. It is unlike the previously

described protocols, where each service has its own set of operations. In addition to

invoking methods on web resources, the protocol allows for content negotiation and

password-style authentication:

Content negotiation: Clients’ requests can include information as to what data

representations they can accept (for example, language or media type), enabling the

server to choose the representation that is the most appropriate for the user.

Authentication: Credentials and challenges are used to support password-style

authentication. On the first attempt to access a password-protected area, the server

reply contains a challenge applicable to the resource. Chapter 11 explains challenges.

When a client receives a challenge, it gets the user to type a name and password and

submits the associated credentials with subsequent requests.

HTTP is implemented over TCP. In the original version of the protocol, each clientserver interaction consisted of the following steps:

• The client requests and the server accepts a connection at the default server port

or at a port specified in the URL.

• The client sends a request message to the server.

• The server sends a reply message to the client.

• The connection is closed.

However, establishing and closing a connection for every request-reply exchange is

expensive, overloading the server and causing too many messages to be sent over the

network. Bearing in mind that browsers generally make multiple requests to the same

server – for example, to get the images in a page just supplied – a later version of the

protocol (HTTP 1.1, see RFC 2616 [Fielding et al. 1999]) uses persistent connections –

connections that remain open over a series of request-reply exchanges between clientSECTION 5.2 REQUEST-REPLY PROTOCOLS 209

and server. A persistent connection can be closed by the client or server at any time by

sending an indication to the other participant. Servers will close a persistent connection

when it has been idle for a period of time. It is possible that a client may receive a

message from the server saying that the connection is closed while it is in the middle of

sending another request or requests. In such cases, the browser will resend the requests

without user involvement, provided that the operations involved are idempotent. For

example, the method GET described below is idempotent. Where non-idempotent

operations are involved, the browser should consult the user as to what to do next.

Requests and replies are marshalled into messages as ASCII text strings, but

resources can be represented as byte sequences and may be compressed. The use of text

in the external data representation has simplified the use of HTTP for application

programmers who work directly with the protocol. In this context, a textual

representation does not add much to the length of the messages.

Data resources are supplied as MIME-like structures in arguments and results.

Multipurpose Internet Mail Extensions (MIME), specified in RFC 2045 [Freed and

Borenstein 1996], is a standard for sending multipart data containing, for example, text,

images and sound in email messages. Data is prefixed with its MIME type so that the

recipient will know how to handle it. A MIME type specifies a type and a subtype, for

example, text/plain, text/html, image/gif or image/jpeg. Clients can also specify the

MIME types that they are willing to accept.

HTTP methods • Each client request specifies the name of a method to be applied to a

resource at the server and the URL of that resource. The reply reports on the status of

the request. Requests and replies may also contain resource data, the contents of a form

or the output of a program resource run on the web server. The methods include the

following:

GET: Requests the resource whose URL is given as its argument. If the URL refers

to data, then the web server replies by returning the data identified by that URL. If

the URL refers to a program, then the web server runs the program and returns its

output to the client. Arguments may be added to the URL; for example, GET can be

used to send the contents of a form to a program as an argument. The GET operation

can be made conditional on the date a resource was last modified. GET can also be

configured to obtain parts of the data.

With GET, all the information for the request is provided in the URL (see, for

example, the query string in Section 1.6).

HEAD: This request is identical to GET, but it does not return any data. However, it

does return all the information about the data, such as the time of last modification,

its type or its size.

Figure 5.6 HTTP Request message

method URL or pathname HTTP version headers message body

GET http://www.dcs.qmul.ac.uk/index.html HTTP/ 1.1210 CHAPTER 5 REMOTE INVOCATION

POST: Specifies the URL of a resource (for example a program) that can deal with

the data supplied in the body of the request. The processing carried out on the data

depends on the function of the program specified in the URL. This method is used

when the action may change data on the server. It is designed to deal with:

• providing a block of data to a data-handling process such as a servlet – for

example, submitting a web form to buy something from a web site;

• posting a message to a mailing list or updating details of members of the list;

• extending a database with an append operation.

PUT: Requests that the data supplied in the request is stored with the given URL as

its identifier, either as a modification of an existing resource or as a new resource.

DELETE: The server deletes the resource identified by the given URL. Servers may

not always allow this operation, in which case the reply indicates failure.

OPTIONS: The server supplies the client with a list of methods it allows to be

applied to the given URL (for example GET, HEAD, PUT) and its special

requirements.

TRACE: The server sends back the request message. Used for diagnostic purposes.

The operations PUT and DELETE are idempotent, but POST is not necessarily so

because it can change the state of a resource. The others are safe operations in that they

do not change anything.

The requests described above may be intercepted by a proxy server (see Section 2.3.1).

The responses to GET and HEAD may be cached by proxy servers.

Message contents • The Request message specifies the name of a method, the URL of

a resource, the protocol version, some headers and an optional message body. Figure 5.6

shows the contents of an HTTP Request message whose method is GET. When the URL

specifies a data resource, the GET method does not have a message body.

Requests to proxies need the absolute URL, as shown in Figure 5.6. Requests to

origin servers (the origin server is where the resource resides) specify a pathname and

give the DNS name of the origin server in a Host header field. For example,

GET /index.html HTTP/1.1

Host: www.dcs.qmul.ac.uk

In general, the header fields contain request modifiers and client information, such as

conditions on the latest date of modification of the resource or acceptable content types

(for example, HTML text, audio or JPEG images). An authorization field can be used to

provide the client’s credentials in the form of a certificate specifying their rights to

access a resource.

A Reply message specifies the protocol version, a status code and ‘reason’, some

headers and an optional message body, as shown in Figure 5.7. The status code and

reason provide a report on the server’s success or otherwise in carrying out the request:

the former is a three-digit integer for interpretation by a program, and the latter is a

textual phrase that can be understood by a person. The header fields are used to pass

additional information about the server or access to the resource. For example, if the

request requires authentication, the status of the response indicates this and a headerSECTION 5.3 REMOTE PROCEDURE CALL 211

field contains a challenge. Some status returns have quite complex effects. In particular,

a 303 status response tells the browser to look under a different URL, which is supplied

in a header field in the reply. It is intended for use in a response from a program activated

by a POST request when the program needs to redirect the browser to a selected

resource.

The message body in request or reply messages contains the data associated with

the URL specified in the request. The message body has its own headers specifying

information about the data, such as its length, its MIME type, its character set, its content

encoding and the last date it was modified. The MIME type field specifies the type of

the data, for example image/jpeg or text/plain. The content encoding field specifies the

compression algorithm to be used

5.3 Remote procedure call

As mentioned in Chapter 2, the concept of a remote procedure call (RPC) represents a

major intellectual breakthrough in distributed computing, with the goal of making the

programming of distributed systems look similar, if not identical, to conventional

programming – that is, achieving a high level of distribution transparency. This

unification is achieved in a very simple manner, by extending the abstraction of a

procedure call to distributed environments. In particular, in RPC, procedures on remote

machines can be called as if they are procedures in the local address space. The

underlying RPC system then hides important aspects of distribution, including the

encoding and decoding of parameters and results, the passing of messages and the

preserving of the required semantics for the procedure call. This concept was first

introduced by Birrell and Nelson [1984] and paved the way for many of the

developments in distributed systems programming used today.

5.3.1 Design issues for RPC

Before looking at the implementation of RPC systems, we look at three issues that are

important in understanding this concept:

• the style of programming promoted by RPC – programming with interfaces;

• the call semantics associated with RPC;

• the key issue of transparency and how it relates to remote procedure calls.

Programming with interfaces • Most modern programming languages provide a means

of organizing a program as a set of modules that can communicate with one another.

Communication between modules can be by means of procedure calls between modules

Figure 5.7 HTTP Reply message

HTTP version status code reason headers message body

HTTP/1.1 200 OK resource data212 CHAPTER 5 REMOTE INVOCATION

or by direct access to the variables in another module. In order to control the possible

interactions between modules, an explicit interface is defined for each module. The

interface of a module specifies the procedures and the variables that can be accessed

from other modules. Modules are implemented so as to hide all the information about

them except that which is available through its interface. So long as its interface remains

the same, the implementation may be changed without affecting the users of the module.

Interfaces in distributed systems: In a distributed program, the modules can run in

separate processes. In the client-server model, in particular, each server provides a set

of procedures that are available for use by clients. For example, a file server would

provide procedures for reading and writing files. The term service interface is used to

refer to the specification of the procedures offered by a server, defining the types of the

arguments of each of the procedures.

There are a number of benefits to programming with interfaces in distributed

systems, stemming from the important separation between interface and

implementation:

• As with any form of modular programming, programmers are concerned only

with the abstraction offered by the service interface and need not be aware of

implementation details.

• Extrapolating to (potentially heterogeneous) distributed systems, programmers

also do not need to know the programming language or underlying platform used

to implement the service (an important step towards managing heterogeneity in

distributed systems).

• This approach provides natural support for software evolution in that

implementations can change as long as long as the interface (the external view)

remains the same. More correctly, the interface can also change as long as it

remains compatible with the original.

The definition of service interfaces is influenced by the distributed nature of the

underlying infrastructure:

• It is not possible for a client module running in one process to access the variables

in a module in another process. Therefore the service interface cannot specify

direct access to variables. Note that CORBA IDL interfaces can specify attributes,

which seems to break this rule. However, the attributes are not accessed directly

but by means of some getter and setter procedures added automatically to the

interface.

• The parameter-passing mechanisms used in local procedure calls – for example,

call by value and call by reference, are not suitable when the caller and procedure

are in different processes. In particular, call by reference is not supported. Rather,

the specification of a procedure in the interface of a module in a distributed

program describes the parameters as input or output, or sometimes both. Input

parameters are passed to the remote server by sending the values of the arguments

in the request message and then supplying them as arguments to the operation to

be executed in the server. Output parameters are returned in the reply message and

are used as the result of the call or to replace the values of the correspondingSECTION 5.3 REMOTE PROCEDURE CALL 213

variables in the calling environment. When a parameter is used for both input and

output, the value must be transmitted in both the request and reply messages.

• Another difference between local and remote modules is that addresses in one

process are not valid in another remote one. Therefore, addresses cannot be passed

as arguments or returned as results of calls to remote modules.

These constraints have a significant impact on the specification of interface definition

languages, as discussed below.

Interface definition languages: An RPC mechanism can be integrated with a particular

programming language if it includes an adequate notation for defining interfaces,

allowing input and output parameters to be mapped onto the language’s normal use of

parameters. This approach is useful when all the parts of a distributed application can be

written in the same language. It is also convenient because it allows the programmer to

use a single language, for example, Java, for local and remote invocation.

However, many existing useful services are written in C++ and other languages.

It would be beneficial to allow programs written in a variety of languages, including

Java, to access them remotely. Interface definition languages (IDLs) are designed to

allow procedures implemented in different languages to invoke one another. An IDL

provides a notation for defining interfaces in which each of the parameters of an

operation may be described as for input or output in addition to having its type specified.

Figure 5.8 shows a simple example of CORBA IDL. The Person structure is the

same as the one used to illustrate marshalling in Section 4.3.1. The interface named

PersonList specifies the methods available for RMI in a remote object that implements

that interface. For example, the method addPerson specifies its argument as in, meaning

that it is an input argument, and the method getPerson that retrieves an instance of

Person by name specifies its second argument as out, meaning that it is an output

argument.

Figure 5.8 CORBA IDL example

// In file Person.idl

struct Person {

string name;

string place;

long year;

};

interface PersonList {

readonly attribute string listname;

void addPerson(in Person p) ;

void getPerson(in string name, out Person p);

long number();

};214 CHAPTER 5 REMOTE INVOCATION

The concept of an IDL was initially developed for RPC systems but applies

equally to RMI and also web services. Our case studies include:

• Sun XDR as an example of an IDL for RPC (in Section 5.3.3);

• CORBA IDL as an example of an IDL for RMI (in Chapter 8 and also included

above);

• the Web Services Description Language (WSDL), which is designed for an

Internet-wide RPC supporting web services (see Section 9.3);

• and protocol buffers used at Google for storing and interchanging many kinds of

structured information (see Section 21.4.1).

RPC call semantics • Request-reply protocols were discussed in Section 5.2, where we

showed that doOperation can be implemented in different ways to provide different

delivery guarantees. The main choices are:

Retry request message: Controls whether to retransmit the request message until

either a reply is received or the server is assumed to have failed.

Duplicate filtering: Controls when retransmissions are used and whether to filter out

duplicate requests at the server.

Retransmission of results: Controls whether to keep a history of result messages to

enable lost results to be retransmitted without re-executing the operations at the

server.

Combinations of these choices lead to a variety of possible semantics for the reliability

of remote invocations as seen by the invoker. Figure 5.9 shows the choices of interest,

with corresponding names for the semantics that they produce. Note that for local

procedure calls, the semantics are exactly once, meaning that every procedure is

executed exactly once (except in the case of process failure). The choices of RPC

invocation semantics are defined as follows.

Figure 5.9 Call semantics

Fault tolerance measures Call semantics

Retransmit request

message

Duplicate

filtering

Re-execute procedure

or retransmit reply

No Not applicable Not applicable Maybe

Yes No Re-execute procedure At-least-once

Yes Yes Retransmit reply At-most-onceSECTION 5.3 REMOTE PROCEDURE CALL 215

Maybe semantics: With maybe semantics, the remote procedure call may be executed

once or not at all. Maybe semantics arises when no fault-tolerance measures are applied

and can suffer from the following types of failure:

• omission failures if the request or result message is lost;

• crash failures when the server containing the remote operation fails.

If the result message has not been received after a timeout and there are no retries, it is

uncertain whether the procedure has been executed. If the request message was lost, then

the procedure will not have been executed. On the other hand, the procedure may have

been executed and the result message lost. A crash failure may occur either before or

after the procedure is executed. Moreover, in an asynchronous system, the result of

executing the procedure may arrive after the timeout. Maybe semantics is useful only for

applications in which occasional failed calls are acceptable.

At-least-once semantics: With at-least-once semantics, the invoker receives either a

result, in which case the invoker knows that the procedure was executed at least once,

or an exception informing it that no result was received. At-least-once semantics can be

achieved by the retransmission of request messages, which masks the omission failures

of the request or result message. At-least-once semantics can suffer from the following

types of failure:

• crash failures when the server containing the remote procedure fails;

• arbitrary failures – in cases when the request message is retransmitted, the remote

server may receive it and execute the procedure more than once, possibly causing

wrong values to be stored or returned.

Section 5.2 defines an idempotent operation as one that can be performed repeatedly

with the same effect as if it had been performed exactly once. Non-idempotent

operations can have the wrong effect if they are performed more than once. For example,

an operation to increase a bank balance by $10 should be performed only once; if it were

to be repeated, the balance would grow and grow! If the operations in a server can be

designed so that all of the procedures in their service interfaces are idempotent

operations, then at-least-once call semantics may be acceptable.

At-most-once semantics: With at-most-once semantics, the caller receives either a

result, in which case the caller knows that the procedure was executed exactly once, or

an exception informing it that no result was received, in which case the procedure will

have been executed either once or not at all. At-most-once semantics can be achieved by

using all of the fault-tolerance measures outlined in Figure 5.9. As in the previous case,

the use of retries masks any omission failures of the request or result messages. This set

of fault tolerance measures prevents arbitrary failures by ensuring that for each RPC a

procedure is never executed more than once. Sun RPC (discussed in Section 5.3.3)

provides at-least-once call semantics.

Transparency • The originators of RPC, Birrell and Nelson [1984], aimed to make

remote procedure calls as much like local procedure calls as possible, with no distinction

in syntax between a local and a remote procedure call. All the necessary calls to

marshalling and message-passing procedures were hidden from the programmer making

the call. Although request messages are retransmitted after a timeout, this is transparent

to the caller to make the semantics of remote procedure calls like that of local procedure

calls.216 CHAPTER 5 REMOTE INVOCATION

More precisely, returning to the terminology of Chapter 1, RPC strives to offer at

least location and access transparency, hiding the physical location of the (potentially

remote) procedure and also accessing local and remote procedures in the same way.

Middleware can also offer additional levels of transparency to RPC.

However, remote procedure calls are more vulnerable to failure than local ones,

since they involve a network, another computer and another process. Whichever of the

above semantics is chosen, there is always the chance that no result will be received, and

in the case of failure, it is impossible to distinguish between failure of the network and

of the remote server process. This requires that clients making remote calls are able to

recover from such situations.

The latency of a remote procedure call is several orders of magnitude greater than

that of a local one. This suggests that programs that make use of remote calls need to be

able to take this factor into account, perhaps by minimizing remote interactions. The

designers of Argus [Liskov and Scheifler 1982] suggested that a caller should be able to

abort a remote procedure call that is taking too long in such a way that it has no effect

on the server. To allow this, the server would need to be able to restore things to how

they were before the procedure was called. These issues are discussed in Chapter 16.

Remote procedure calls also require a different style of parameter passing, as

discussed above. In particular, RPC does not offer call by reference.

Waldo et al. [1994] say that the difference between local and remote operations

should be expressed at the service interface, to allow participants to react in a consistent

way to possible partial failures. Other systems went further than this by arguing that the

syntax of a remote call should be different from that of a local call: in the case of Argus,

the language was extended to make remote operations explicit to the programmer.

The choice as to whether RPC should be transparent is also available to the

designers of IDLs. For example, in some IDLs, a remote invocation may throw an

exception when the client is unable to communicate with a remote procedure. This

requires that the client program handle such exceptions, allowing it to deal with such

failures. An IDL can also provide a facility for specifying the call semantics of a

procedure. This can help the designer of the service – for example, if at-least-once call

semantics is chosen to avoid the overheads of at-most-once, the operations must be

designed to be idempotent.

The current consensus is that remote calls should be made transparent in the sense

that the syntax of a remote call is the same as that of a local invocation, but that the

difference between local and remote calls should be expressed in their interfaces.

5.3.2 Implementation of RPC

The software components required to implement RPC are shown in Figure 5.10. The

client that accesses a service includes one stub procedure for each procedure in the

service interface. The stub procedure behaves like a local procedure to the client, but

instead of executing the call, it marshals the procedure identifier and the arguments into

a request message, which it sends via its communication module to the server. When the

reply message arrives, it unmarshals the results. The server process contains a dispatcher

together with one server stub procedure and one service procedure for each procedure

in the service interface. The dispatcher selects one of the server stub procedures

according to the procedure identifier in the request message. The server stub procedureSECTION 5.3 REMOTE PROCEDURE CALL 217

then unmarshals the arguments in the request message, calls the corresponding service

procedure and marshals the return values for the reply message. The service procedures

implement the procedures in the service interface. The client and server stub procedures

and the dispatcher can be generated automatically by an interface compiler from the

interface definition of the service.

RPC is generally implemented over a request-reply protocol like the ones

discussed in Section 5.2. The contents of request and reply messages are the same as

those illustrated for request-reply protocols in Figure 5.4. RPC may be implemented to

have one of the choices of invocation semantics discussed in Section 5.3.1 – at-leastonce or at-most-once is generally chosen. To achieve this, the communication module

will implement the desired design choices in terms of retransmission of requests, dealing

with duplicates and retransmission of results, as shown in Figure 5.9.

5.3.3 Case study: Sun RPC

RFC 1831 [Srinivasan 1995a] describes Sun RPC, which was designed for client-server

communication in the Sun Network File System (NFS). Sun RPC is sometimes called

ONC (Open Network Computing) RPC. It is supplied as a part of the various Sun and

other UNIX operating systems and is also available with NFS installations.

Implementors have the choice of using remote procedure calls over either UDP or TCP.

When Sun RPC is used with UDP, request and reply messages are restricted in length –

theoretically to 64 kilobytes, but more often in practice to 8 or 9 kilobytes. It uses atleast-once call semantics. Broadcast RPC is an option.

The Sun RPC system provides an interface language called XDR and an interface

compiler called rpcgen, which is intended for use with the C programming language.

Interface definition language • The Sun XDR language, which was originally designed

for specifying external data representations, was extended to become an interface

definition language. It may be used to define a service interface for Sun RPC by

specifying a set of procedure definitions together with supporting type definitions. The

notation is rather primitive in comparison with that used by CORBA IDL or Java. In

particular:

Figure 5.10 Role of client and server stub procedures in RPC

client

Request

Reply

Communication Communication

module module dispatcher

service

client stub server stub

procedure procedure

client process server process

program procedure218 CHAPTER 5 REMOTE INVOCATION

• Most languages allow interface names to be specified, but Sun RPC does not –

instead of this, a program number and a version number are supplied. The program

numbers can be obtained from a central authority to allow every program to have

its own unique number. The version number is intended to be changed when a

procedure signature changes. Both program and version number are passed in the

request message, so the client and server can check that they are using the same

version.

• A procedure definition specifies a procedure signature and a procedure number.

The procedure number is used as a procedure identifier in request messages.

• Only a single input parameter is allowed. Therefore, procedures requiring

multiple parameters must include them as components of a single structure.

• The output parameters of a procedure are returned via a single result.

• The procedure signature consists of the result type, the name of the procedure and

the type of the input parameter. The type of both the result and the input parameter

may specify either a single value or a structure containing several values.

Figure 5.11 Files interface in Sun XDR

const MAX = 1000;

typedef int FileIdentifier;

typedef int FilePointer;

typedef int Length;

struct Data {

int length;

char buffer[MAX];

};

struct writeargs {

FileIdentifier f;

FilePointer position;

Data data;

};

struct readargs {

FileIdentifier f;

FilePointer position;

Length length;

};

program FILEREADWRITE {

version VERSION {

void WRITE(writeargs)=1; 1

Data READ(readargs)=2; 2

}=2;

} = 9999;SECTION 5.3 REMOTE PROCEDURE CALL 219

For example, see the XDR definition in Figure 5.11 of an interface with a pair of

procedures for writing and reading files. The program number is 9999 and the version

number is 2. The READ procedure (line 2) takes as its input parameter a structure with

three components specifying a file identifier, a position in the file and the number of

bytes required. Its result is a structure containing the number of bytes returned and the

file data. The WRITE procedure (line 1) has no result. The WRITE and READ procedures

are given numbers 1 and 2. The number 0 is reserved for a null procedure, which is

generated automatically and is intended to be used to test whether a server is available.

This interface definition language provides a notation for defining constants,

typedefs, structures, enumerated types, unions and programs. Typedefs, structures and

enumerated types use the C language syntax. The interface compiler rpcgen can be used

to generate the following from an interface definition:

• client stub procedures;

• server main procedure, dispatcher and server stub procedures;

• XDR marshalling and unmarshalling procedures for use by the dispatcher and

client and server stub procedures.

Binding • Sun RPC runs a local binding service called the port mapper at a well-known

port number on each computer. Each instance of a port mapper records the program

number, version number and port number in use by each service running locally. When

a server starts up it registers its program number, version number and port number with

the local port mapper. When a client starts up, it finds out the server’s port by making a

remote request to the port mapper at the server’s host, specifying the program number

and version number.

When a service has multiple instances running on different computers, the

instances may use different port numbers for receiving client requests. If a client needs

to multicast a request to all the instances of a service that are using different port

numbers, it cannot use a direct IP multicast message for this purpose. The solution is that

clients make multicast remote procedure calls by multicasting them to all the port

mappers, specifying the program and version number. Each port mapper forwards all

such calls to the appropriate local service program, if there is one.

Authentication. Sun RPC request and reply messages provide additional fields enabling

authentication information to be passed between client and server. The request message

contains the credentials of the user running the client program. For example, in the

UNIX style of authentication the credentials include the uid and gid of the user. Access

control mechanisms can be built on top of the authentication information which is made

available to the server procedures via a second argument. The server program is

responsible for enforcing access control by deciding whether to execute each procedure

call according to the authentication information. For example, if the server is an NFS file

server, it can check whether the user has sufficient rights to carry out a requested file

operation. Several different authentication protocols can be supported. These include:

• none;

• UNIX style, as described above;

• a style in which a shared key is established for signing the RPC messages;

• Kerberos (see Chapter 11).

A field in the RPC header indicates which style is being used.220 CHAPTER 5 REMOTE INVOCATION

A more generic approach to security is described in RFC 2203 [Eisler et al. 1997].

It provides for the secrecy and integrity of RPC messages as well as authentication. It

allows the client and server to negotiate a security context in which either no security is

applied, or in the case that security is required, message integrity or message privacy or

both may be applied.

Client and server programs • Further material on Sun RPC is available at

www.cdk5.net/rmi. It includes example client and server programs corresponding to the

interface defined in Figure 5.11.

5.4 Remote method invocation

Remote method invocation (RMI) is closely related to RPC but extended into the world

of distributed objects. In RMI, a calling object can invoke a method in a potentially

remote object. As with RPC, the underlying details are generally hidden from the user.

The commonalities between RMI and RPC are as follows:

• They both support programming with interfaces, with the resultant benefits that

stem from this approach (see Section 5.3.1).

• They are both typically constructed on top of request-reply protocols and can offer

a range of call semantics such as at-least-once and at-most-once.

• They both offer a similar level of transparency – that is, local and remote calls

employ the same syntax but remote interfaces typically expose the distributed

nature of the underlying call, for example by supporting remote exceptions.

The following differences lead to added expressiveness when it comes to the

programming of complex distributed applications and services.

• The programmer is able to use the full expressive power of object-oriented

programming in the development of distributed systems software, including the

use of objects, classes and inheritance, and can also employ related objectoriented design methodologies and associated tools.

• Building on the concept of object identity in object-oriented systems, all objects

in an RMI-based system have unique object references (whether they are local or

remote), such object references can also be passed as parameters, thus offering

significantly richer parameter-passing semantics than in RPC.

The issue of parameter passing is particularly important in distributed systems. RMI

allows the programmer to pass parameters not only by value, as input or output

parameters, but also by object reference. Passing references is particularly attractive if

the underlying parameter is large or complex. The remote end, on receiving an object

reference, can then access this object using remote method invocation, instead of having

to transmit the object value across the network.

The rest of this section examines the concept of remote method invocation in more

detail, looking initially at the key issues surrounding distributed object models before

looking at implementation issues surrounding RMI, including distributed garbage

collection.SECTION 5.4 REMOTE METHOD INVOCATION 221

5.4.1 Design issues for RMI

As mentioned above, RMI shares the same design issues as RPC in terms of

programming with interfaces, call semantics and level of transparency. The reader is

encouraged to refer back to Section 5.3.1 for discussion of these items.

The key added design issue relates to the object model and, in particular,

achieving the transition from objects to distributed objects. We first describe the

conventional, single-image object model and then describe the distributed object model.

The object model • An object-oriented program, for example in Java or C++, consists

of a collection of interacting objects, each of which consists of a set of data and a set of

methods. An object communicates with other objects by invoking their methods,

generally passing arguments and receiving results. Objects can encapsulate their data

and the code of their methods. Some languages, for example Java and C++, allow

programmers to define objects whose instance variables can be accessed directly. But

for use in a distributed object system, an object’s data should be accessible only via its

methods.

Object references: Objects can be accessed via object references. For example, in Java, a

variable that appears to hold an object actually holds a reference to that object. To

invoke a method in an object, the object reference and method name are given, together

with any necessary arguments. The object whose method is invoked is sometimes called

the target and sometimes the receiver. Object references are first-class values, meaning

that they may, for example, be assigned to variables, passed as arguments and returned

as results of methods.

Interfaces: An interface provides a definition of the signatures of a set of methods (that

is, the types of their arguments, return values and exceptions) without specifying their

implementation. An object will provide a particular interface if its class contains code

that implements the methods of that interface. In Java, a class may implement several

interfaces, and the methods of an interface may be implemented by any class. An

interface also defines types that can be used to declare the type of variables or of the

parameters and return values of methods. Note that interfaces do not have constructors.

Actions : Action in an object-oriented program is initiated by an object invoking a

method in another object. An invocation can include additional information (arguments)

needed to carry out the method. The receiver executes the appropriate method and then

returns control to the invoking object, sometimes supplying a result. An invocation of a

method can have three effects:

1. The state of the receiver may be changed.

2. A new object may be instantiated, for example, by using a constructor in Java or

C++.

3. Further invocations on methods in other objects may take place.

As an invocation can lead to further invocations of methods in other objects, an action

is a chain of related method invocations, each of which eventually returns.

Exceptions: Programs can encounter many sorts of errors and unexpected conditions of

varying seriousness. During the execution of a method, many different problems may be

discovered: for example, inconsistent values in the object’s variables, or failures in222 CHAPTER 5 REMOTE INVOCATION

attempts to read or write to files or network sockets. When programmers need to insert

tests in their code to deal with all possible unusual or erroneous cases, this detracts from

the clarity of the normal case. Exceptions provide a clean way to deal with error

conditions without complicating the code. In addition, each method heading explicitly

lists as exceptions the error conditions it might encounter, allowing users of the method

to deal with them. A block of code may be defined to throw an exception whenever

particular unexpected conditions or errors arise. This means that control passes to

another block of code that catches the exception. Control does not return to the place

where the exception was thrown.

Garbage collection: It is necessary to provide a means of freeing the space occupied by

objects when they are no longer needed. A language such as Java, that can detect

automatically when an object is no longer accessible recovers the space and makes it

available for allocation to other objects. This process is called garbage collection. When

a language (for example, C++) does not support garbage collection, the programmer has

to cope with the freeing of space allocated to objects. This can be a major source of

errors.

Distributed objects • The state of an object consists of the values of its instance

variables. In the object-based paradigm the state of a program is partitioned into separate

parts, each of which is associated with an object. Since object-based programs are

logically partitioned, the physical distribution of objects into different processes or

computers in a distributed system is a natural extension (the issue of placement is

discussed in Section 2.3.1).

Distributed object systems may adopt the client-server architecture. In this case,

objects are managed by servers and their clients invoke their methods using remote

method invocation. In RMI, the client’s request to invoke a method of an object is sent

in a message to the server managing the object. The invocation is carried out by

executing a method of the object at the server and the result is returned to the client in

another message. To allow for chains of related invocations, objects in servers are

allowed to become clients of objects in other servers.

Distributed objects can assume other architectural models. For example, objects

can be replicated in order to obtain the usual benefits of fault tolerance and enhanced

performance, and objects can be migrated with a view to enhancing their performance

and availability.

Having client and server objects in different processes enforces encapsulation.

That is, the state of an object can be accessed only by the methods of the object, which

means that it is not possible for unauthorized methods to act on the state. For example,

the possibility of concurrent RMIs from objects in different computers implies that an

object may be accessed concurrently. Therefore the possibility of conflicting accesses

arises. However, the fact that the data of an object is accessed only by its own methods

allows objects to provide methods for protecting themselves against incorrect accesses.

For example, they may use synchronization primitives such as condition variables to

protect access to their instance variables.

Another advantage of treating the shared state of a distributed program as a

collection of objects is that an object may be accessed via RMI, or it may be copied into

a local cache and accessed directly, provided that the class implementation is available

locally.SECTION 5.4 REMOTE METHOD INVOCATION 223

The fact that objects are accessed only via their methods gives rise to another

advantage of heterogeneous systems, that different data formats may be used at different

sites – these formats will be unnoticed by clients that use RMI to access the methods of

the objects.

The distributed object model • This section discusses extensions to the object model to

make it applicable to distributed objects. Each process contains a collection of objects,

some of which can receive both local and remote invocations, whereas the other objects

can receive only local invocations, as shown in Figure 5.12. Method invocations

between objects in different processes, whether in the same computer or not, are known

as remote method invocations. Method invocations between objects in the same process

are local method invocations.

We refer to objects that can receive remote invocations as remote objects. In

Figure 5.12, the objects B and F are remote objects. All objects can receive local

invocations, although they can receive them only from other objects that hold references

to them. For example, object C must have a reference to object E so that it can invoke

one of its methods. The following two fundamental concepts are at the heart of the

distributed object model:

Remote object references: Other objects can invoke the methods of a remote object

if they have access to its remote object reference. For example, a remote object

reference for B in Figure 5.12 must be available to A.

Remote interfaces: Every remote object has a remote interface that specifies which

of its methods can be invoked remotely. For example, the objects B and F in Figure

5.12 must have remote interfaces.

We look at remote object references, remote interfaces and other aspects of the

distributed object model next.

Remote object references: The notion of object reference is extended to allow any object

that can receive an RMI to have a remote object reference. A remote object reference is

an identifier that can be used throughout a distributed system to refer to a particular

unique remote object. Its representation, which is generally different from that of a local

object reference is discussed in Section 4.3.4. Remote object references are analogous

to local ones in that:

1. The remote object to receive a remote method invocation is specified by the

invoker as a remote object reference.

2. Remote object references may be passed as arguments and results of remote

method invocations.

Figure 5.12 Remote and local method invocations

invocation invoca

tion

remote loca

l

local

local

invocation

invocation

B

C

D

E

invocation

remote

F

A224 CHAPTER 5 REMOTE INVOCATION

Remote interfaces: The class of a remote object implements the methods of its remote

interface, for example as public instance methods in Java. Objects in other processes can

invoke only the methods that belong to its remote interface, as shown in Figure 5.13.

Local objects can invoke the methods in the remote interface as well as other methods

implemented by a remote object. Note that remote interfaces, like all interfaces, do not

have constructors.

The CORBA system provides an interface definition language (IDL), which is

used for defining remote interfaces. See Figure 5.8 for an example of a remote interface

defined in CORBA IDL. The classes of remote objects and the client programs may be

implemented in any language for which an IDL compiler is available, such as C++, Java

or Python. CORBA clients need not use the same language as the remote object in order

to invoke its methods remotely.

In Java RMI, remote interfaces are defined in the same way as any other Java

interface. They acquire their ability to be remote interfaces by extending an interface

named Remote. Both CORBA IDL (Chapter 8) and Java support multiple inheritance of

interfaces. That is, an interface is allowed to extend one or more other interfaces.

Actions in a distributed object system • As in the non-distributed case, an action is

initiated by a method invocation, which may result in further invocations on methods in

other objects. But in the distributed case, the objects involved in a chain of related

invocations may be located in different processes or different computers. When an

invocation crosses the boundary of a process or computer, RMI is used, and the remote

reference of the object must be available to the invoker. In Figure 5.12, object A needs

to hold a remote object reference to object B. Remote object references may be obtained

as the results of remote method invocations. For example, object A in Figure 5.12 might

obtain a remote reference to object F from object B.

When an action leads to the instantiation of a new object, that object will normally

live within the process where instantiation is requested – for example, where the

constructor was used. If the newly instantiated object has a remote interface, it will be a

remote object with a remote object reference.

Distributed applications may provide remote objects with methods for

instantiating objects that can be accessed by RMI, thus effectively providing the effect

of remote instantiation of objects. For example, if the object L in Figure 5.14 contains a

method for creating remote objects, then the remote invocations from C and K could

lead to the instantiation of the objects M and N, respectively.

Figure 5.13 A remote object and its remote interface

interface

remote

m1

m2

m3

m4

m5

m6

Data

implementation

remoteobject

{ of methodsSECTION 5.4 REMOTE METHOD INVOCATION 225

Garbage collection in a distributed-object system: If a language, for example Java,

supports garbage collection, then any associated RMI system should allow garbage

collection of remote objects. Distributed garbage collection is generally achieved by

cooperation between the existing local garbage collector and an added module that

carries out a form of distributed garbage collection, usually based on reference counting.

Section 5.4.3 describes such a scheme in detail. If garbage collection is not available,

then remote objects that are no longer required should be deleted.

Exceptions: Any remote invocation may fail for reasons related to the invoked object

being in a different process or computer from the invoker. For example, the process

containing the remote object may have crashed or may be too busy to reply, or the

invocation or result message may be lost. Therefore, remote method invocation should

be able to raise exceptions such as timeouts that are due to distribution as well as those

raised during the execution of the method invoked. Examples of the latter are an attempt

to read beyond the end of a file, or to access a file without the correct permissions.

CORBA IDL provides a notation for specifying application-level exceptions, and

the underlying system generates standard exceptions when errors due to distribution

occur. CORBA client programs need to be able to handle exceptions. For example, a

C++ client program will use the exception mechanisms in C++.

5.4.2 Implementation of RMI

Several separate objects and modules are involved in achieving a remote method

invocation. These are shown in Figure 5.15, in which an application-level object A

invokes a method in a remote application-level object B for which it holds a remote

object reference. This section discusses the roles of each of the components shown in

that figure, dealing first with the communication and remote reference modules and then

with the RMI software that runs over them.

We then explore the following related topics: the generation of proxies, the

binding of names to their remote object references, the activation and passivation of

objects and the location of objects from their remote object references.

Communication module • The two cooperating communication modules carry out the

request-reply protocol, which transmits request and reply messages between the client

and server. The contents of request and reply messages are shown in Figure 5.4. The

communication module uses only the first three items, which specify the message type,

its requestId and the remote reference of the object to be invoked. The operationId and

Figure 5.14 Instantiation of remote objects

C

M N

K

invocation

remote

invocation

remote

L

instantiate instantiate226 CHAPTER 5 REMOTE INVOCATION

all the marshalling and unmarshalling are the concern of the RMI software, discussed

below. The communication modules are together responsible for providing a specified

invocation semantics, for example at-most-once.

The communication module in the server selects the dispatcher for the class of the

object to be invoked, passing on its local reference, which it gets from the remote

reference module in return for the remote object identifier in the request message. The

role of dispatcher is discussed in the forthcoming section on RMI software.

Remote reference module • A remote reference module is responsible for translating

between local and remote object references and for creating remote object references.

To support its responsibilities, the remote reference module in each process has a remote

object table that records the correspondence between local object references in that

process and remote object references (which are system-wide). The table includes:

• An entry for all the remote objects held by the process. For example, in Figure

5.15 the remote object B will be recorded in the table at the server.

• An entry for each local proxy. For example, in Figure 5.15 the proxy for B will be

recorded in the table at the client.

The role of a proxy is discussed in the subsection on RMI software. The actions of the

remote reference module are as follows:

• When a remote object is to be passed as an argument or a result for the first time,

the remote reference module is asked to create a remote object reference, which it

adds to its table.

• When a remote object reference arrives in a request or reply message, the remote

reference module is asked for the corresponding local object reference, which may

refer either to a proxy or to a remote object. In the case that the remote object

reference is not in the table, the RMI software creates a new proxy and asks the

remote reference module to add it to the table.

This module is called by components of the RMI software when they are marshalling

and unmarshalling remote object references. For example, when a request message

arrives, the table is used to find out which local object is to be invoked.

Figure 5.15 The role of proxy and skeleton in remote method invocation

object A proxy for B Request skeleton object B

Reply

Remote Communication Communication Remote reference

reference module module module module

for B’s class

& dispatcher

client server remote

servantSECTION 5.4 REMOTE METHOD INVOCATION 227

Servants • A servant is an instance of a class that provides the body of a remote object.

It is the servant that eventually handles the remote requests passed on by the

corresponding skeleton. Servants live within a server process. They are created when

remote objects are instantiated and remain in use until they are no longer needed, finally

being garbage collected or deleted.

The RMI software • This consists of a layer of software between the application-level

objects and the communication and remote reference modules. The roles of the

middleware objects shown in Figure 5.15 are as follows:

Proxy: The role of a proxy is to make remote method invocation transparent to

clients by behaving like a local object to the invoker; but instead of executing an

invocation, it forwards it in a message to a remote object. It hides the details of the

remote object reference, the marshalling of arguments, unmarshalling of results and

sending and receiving of messages from the client. There is one proxy for each

remote object for which a process holds a remote object reference. The class of a

proxy implements the methods in the remote interface of the remote object it

represents, which ensures that remote method invocations are suitable for the type of

the remote object. However, the proxy implements them quite differently. Each

method of the proxy marshals a reference to the target object, its own operationId and

its arguments into a request message and sends it to the target. It then awaits the reply

message, unmarshals it and returns the results to the invoker.

Dispatcher: A server has one dispatcher and one skeleton for each class

representing a remote object. In our example, the server has a dispatcher and a

skeleton for the class of remote object B. The dispatcher receives request messages

from the communication module. It uses the operationId to select the appropriate

method in the skeleton, passing on the request message. The dispatcher and the proxy

use the same allocation of operationIds to the methods of the remote interface.

Skeleton: The class of a remote object has a skeleton, which implements the methods

in the remote interface. They are implemented quite differently from the methods in

the servant that incarnates a remote object. A skeleton method unmarshals the

arguments in the request message and invokes the corresponding method in the

servant. It waits for the invocation to complete and then marshals the result, together

with any exceptions, in a reply message to the sending proxy’s method.

Remote object references are marshalled in the form shown in Figure 4.13, which

includes information about the remote interface of the remote object – for example, the

name of the remote interface or the class of the remote object. This information enables

the proxy class to be determined so that a new proxy may be created when it is needed.

For example, the proxy class name may be generated by appending \_proxy to the name

of the remote interface.

Generation of the classes for proxies, dispatchers and skeletons • The classes for the

proxy, dispatcher and skeleton used in RMI are generated automatically by an interface

compiler. For example, in the Orbix implementation of CORBA, interfaces of remote

objects are defined in CORBA IDL, and the interface compiler can be used to generate

the classes for proxies, dispatchers and skeletons in C++ or in Java [www.iona.com].

For Java RMI, the set of methods offered by a remote object is defined as a Java interface228 CHAPTER 5 REMOTE INVOCATION

that is implemented within the class of the remote object. The Java RMI compiler

generates the proxy, dispatcher and skeleton classes from the class of the remote object.

Dynamic invocation: An alternative to proxies • The proxy just described is static, in the

sense that its class is generated from an interface definition and then compiled into the

client code. Sometimes this is not practical, though. Suppose that a client program

receives a remote reference to an object whose remote interface was not available at

compile time. In this case it needs another way to invoke the remote object. Dynamic

invocation gives the client access to a generic representation of a remote invocation like

the doOperation method used in Exercise 5.18, which is available as part of the

infrastructure for RMI (see Section 5.4.1). The client will supply the remote object

reference, the name of the method and the arguments to doOperation and then wait to

receive the results.

Note that although the remote object reference includes information about the

interface of the remote object, such as its name, this is not enough – the names of the

methods and the types of the arguments are required for making a dynamic invocation.

CORBA provides this information via a component called the Interface Repository,

which is described in Chapter 8.

The dynamic invocation interface is not as convenient to use as a proxy, but it is

useful in applications where some of the interfaces of the remote objects cannot be

predicted at design time. An example of such an application is the shared whiteboard

that we use to illustrate Java RMI (Section 5.5), CORBA (Chapter 8) and web services

(Section 9.2.3). To summarize: the shared whiteboard application displays many

different types of shapes, such as circles, rectangles and lines, but it should also be able

to display new shapes that were not predicted when the client was compiled. A client

that uses dynamic invocation is able to address this challenge. We shall see in Section

5.5 that the dynamic downloading of classes to clients is an alternative to dynamic

invocation. This is available in Java RMI – a single-language system.

Dynamic skeletons: It is clear, from the above example, that it can also arise that a server

will need to host remote objects whose interfaces were not known at compile time. For

example, a client may supply a new type of shape to the shared whiteboard server for it

to store. A server with dynamic skeletons would be able to deal with this situation. We

defer describing dynamic skeletons until the chapter on CORBA (Chapter 8). However,

as we shall see in Section 5.5, Java RMI addresses this problem by using a generic

dispatcher and the dynamic downloading of classes to the server.

Server and client programs • The server program contains the classes for the

dispatchers and skeletons, together with the implementations of the classes of all of the

servants that it supports. In addition, the server program contains an initialization

section (for example, in a main method in Java or C++). The initialization section is

responsible for creating and initializing at least one of the servants to be hosted by the

server. Additional servants may be created in response to requests from clients. The

initialization section may also register some of its servants with a binder (see below).

Generally it will register just one servant, which can be used to access the rest.

The client program will contain the classes of the proxies for all of the remote

objects that it will invoke. It can use a binder to look up remote object references.SECTION 5.4 REMOTE METHOD INVOCATION 229

Factory methods: We noted earlier that remote object interfaces cannot include

constructors. This means that servants cannot be created by remote invocation on

constructors. Servants are created either in the initialization section or in methods in a

remote interface designed for that purpose. The term factory method is sometimes used

to refer to a method that creates servants, and a factory object is an object with factory

methods. Any remote object that needs to be able to create new remote objects on

demand for clients must provide methods in its remote interface for this purpose. Such

methods are called factory methods, although they are really just normal methods.

The binder • Client programs generally require a means of obtaining a remote object

reference for at least one of the remote objects held by a server. For example, in Figure

5.12, object A would require a remote object reference for object B. A binder in a

distributed system is a separate service that maintains a table containing mappings from

textual names to remote object references. It is used by servers to register their remote

objects by name and by clients to look them up. Chapter 8 contains a discussion of the

CORBA Naming Service. The Java binder, RMIregistry, is discussed briefly in the case

study on Java RMI in Section 5.5.

Server threads • Whenever an object executes a remote invocation, that execution may

lead to further invocations of methods in other remote objects, which may take some

time to return. To avoid the execution of one remote invocation delaying the execution

of another, servers generally allocate a separate thread for the execution of each remote

invocation. When this is the case, the designer of the implementation of a remote object

must allow for the effects on its state of concurrent executions.

Activation of remote objects • Some applications require that information survive for

long periods of time. However, it is not practical for the objects representing such

information to be kept in running processes for unlimited periods, particularly since they

are not necessarily in use all of the time. To avoid the potential waste of resources that

would result from to running all of the servers that manage remote objects all of the time,

the servers can be started whenever they are needed by clients, as is done for the standard

set of TCP services such as FTP, which are started on demand by a service called Inetd.

Processes that start server processes to host remote objects are called activators, for the

following reasons.

A remote object is described as active when it is available for invocation within

a running process, whereas it is called passive if is not currently active but can be made

active. A passive object consists of two parts:

1. the implementation of its methods;

2. its state in the marshalled form.

Activation consists of creating an active object from the corresponding passive object by

creating a new instance of its class and initializing its instance variables from the stored

state. Passive objects can be activated on demand, for example when they need to be

invoked by other objects.

An activator is responsible for:

• registering passive objects that are available for activation, which involves

recording the names of servers against the URLs or file names of the

corresponding passive objects;230 CHAPTER 5 REMOTE INVOCATION

• starting named server processes and activating remote objects in them;

• keeping track of the locations of the servers for remote objects that it has already

activated.

Java RMI provides the ability to make some remote objects activatable [java.sun.com

IX]. When an activatable object is invoked, if that object is not currently active, the

object is made active from its marshalled state and then passed the invocation. It uses

one activator on each server computer.

The CORBA case study in Chapter 8 describes the implementation repository – a

weak form of activator that starts services containing objects in an initial state.

Persistent object stores • An object that is guaranteed to live between activations of

processes is called a persistent object. Persistent objects are generally managed by

persistent object stores, which store their state in a marshalled form on disk. Examples

include the CORBA persistent state service (see Chapter 8), Java Data Objects

[java.sun.com VIII] and Persistent Java [Jordan 1996; java.sun.com IV].

In general, a persistent object store will manage very large numbers of persistent

objects, which are stored on disk or in a database until they are needed. They will be

activated when their methods are invoked by other objects. Activation is generally

designed to be transparent – that is, the invoker should not be able to tell whether an

object is already in main memory or has to be activated before its method is invoked.

Persistent objects that are no longer needed in main memory can be passivated. In most

cases, objects are saved in the persistent object store whenever they reach a consistent

state, for the sake of fault tolerance. The persistent object store needs a strategy for

deciding when to passivate objects. For example, it may do so in response to a request

in the program that activated the objects, either at the end of a transaction or when the

program exits. Persistent object stores generally attempt to optimize passivation by

saving only those objects that have been modified since the last time they were saved.

Persistent object stores generally allow collections of related persistent objects to

have human-readable names such as pathnames or URLs. In practice, each humanreadable name is associated with the root of a connected set of persistent objects.

There are two approaches to deciding whether an object is persistent or not:

• The persistent object store maintains some persistent roots, and any object that is

reachable from a persistent root is defined to be persistent. This approach is used

by Persistent Java, Java Data Objects and PerDiS [Ferreira et al. 2000]. They

make use of a garbage collector to dispose of objects that are no longer reachable

from the persistent roots.

• The persistent object store provides some classes on which persistence is based –

persistent objects belong to their subclasses. For example, in Arjuna [Parrington

et al. 1995], persistent objects are based on C++ classes that provide transactions

and recovery. Unwanted objects must be deleted explicitly.

Some persistent object stores, such as PerDiS and Khazana [Carter et al. 1998], allow

objects to be activated in multiple caches local to users, instead of in servers. In this case,

a cache consistency protocol is required. Further details on consistency models can be

found on the companion web site, in the chapter from the fourth edition on distributed

shared memory [www.cdk5.net/dsm].SECTION 5.4 REMOTE METHOD INVOCATION 231

Object location • Section 4.3.4 describes a form of remote object reference that contains

the Internet address and port number of the process that created the remote object as a

way of guaranteeing uniqueness. This form of remote object reference can also be used

as an address for a remote object, so long as that object remains in the same process for

the rest of its life. But some remote objects will exist in a series of different processes,

possibly on different computers, throughout their lifetime. In this case, a remote object

reference cannot act as an address. Clients making invocations require both a remote

object reference and an address to which to send invocations.

A location service helps clients to locate remote objects from their remote object

references. It uses a database that maps remote object references to their probable

current locations – the locations are probable because an object may have migrated again

since it was last heard of. For example, the Clouds system [Dasgupta et al. 1991] and

the Emerald system [Jul et al. 1988] used a cache/broadcast scheme in which a member

of a location service on each computer holds a small cache of remote object referenceto-location mappings. If a remote object reference is in the cache, that address is tried

for the invocation and will fail if the object has moved. To locate an object that has

moved or whose location is not in the cache, the system broadcasts a request. This

scheme may be enhanced by the use of forward location pointers, which contain hints as

to the new location of an object. Another example is the resolution service required for

resolving the URN of a resource into its current URL, mentioned in Section 9.1.

5.4.3 Distributed garbage collection

The aim of a distributed garbage collector is to ensure that if a local or remote reference

to an object is still held anywhere in a set of distributed objects, the object itself will

continue to exist, but as soon as no object any longer holds a reference to it, the object

will be collected and the memory it uses recovered.

We describe the Java distributed garbage collection algorithm, which is similar to

the one described by Birrell et al. [1995]. It is based on reference counting. Whenever a

remote object reference enters a process, a proxy will be created and will stay there for

as long as it is needed. The process where the object lives (its server) should be informed

of the new proxy at the client. Then later when there is no longer a proxy at the client,

the server should be informed. The distributed garbage collector works in cooperation

with the local garbage collectors as follows:

• Each server process maintains a set of the names of the processes that hold remote

object references for each of its remote objects; for example, B.holders is the set

of client processes (virtual machines) that have proxies for object B. (In Figure

5.15, this set will include the client process illustrated.) This set can be held in an

additional column in the remote object table.

• When a client C first receives a remote reference to a particular remote object, B,

it makes an addRef(B) invocation to the server of that remote object and then

creates a proxy; the server adds C to B.holders.232 CHAPTER 5 REMOTE INVOCATION

• When a client C’s garbage collector notices that a proxy for remote object B is no

longer reachable, it makes a removeRef(B) invocation to the corresponding server

and then deletes the proxy; the server removes C from B.holders.

• When B.holders is empty, the server’s local garbage collector will reclaim the

space occupied by B unless there are any local holders.

This algorithm is intended to be carried out by means of pairwise request-reply

communication with at-most-once invocation semantics between the remote reference

modules in processes – it does not require any global synchronization. Note also that the

extra invocations made on behalf of the garbage collection algorithm do not affect every

normal RMI; they occur only when proxies are created and deleted.

There is a possibility that one client may make a removeRef(B) invocation at about

the same time as another client makes an addRef(B) invocation. If the removeRef arrives

first and B.holders is empty, the remote object B could be deleted before the addRef

arrives. To avoid this situation, if the set B.holders is empty at the time when a remote

object reference is transmitted, a temporary entry is added until the addRef arrives.

The Java distributed garbage collection algorithm tolerates communication

failures by using the following approach. The addRef and removeRef operations are

idempotent. In the case that an addRef(B) call returns an exception (meaning that the

method was either executed once or not at all), the client will not create a proxy but will

make a removeRef(B) call. The effect of removeRef is correct whether or not the addRef

succeeded. The case where removeRef fails is dealt with by leases.

The Java distributed garbage collection algorithm can tolerate the failure of client

processes. To achieve this, servers lease their objects to clients for a limited period of

time. The lease period starts when the client makes an addRef invocation to the server.

It ends either when the time has expired or when the client makes a removeRef

invocation to the server. The information stored by the server concerning each lease

contains the identifier of the client’s virtual machine and the period of the lease. Clients

are responsible for requesting the server to renew their leases before they expire.

Leases in Jini • The Jini distributed system includes a specification for leases [Arnold

et al. 1999] that can be used in a variety of situations when one object offers a resource

to another object – for example, when remote objects offer references to other objects.

Objects that offer such resources are at risk of having to maintain the resources when the

users are no longer interested or their programs have exited. To avoid complicated

protocols to discover whether the resource users are still interested, the resources are

offered for a limited period of time. The granting of the use of a resource for a period of

time is called a lease. The object offering the resource will maintain it until the time in

the lease expires. The resource users are responsible for requesting their renewal when

they expire.

The period of a lease may be negotiated between the grantor and the recipient in

Jini, although this does not happen with the leases used in Java RMI. In Jini, an object

representing a lease implements the Lease interface. It contains information about the

period of the lease and methods enabling the lease to be renewed or cancelled. The

grantor returns an instance of a Lease when it supplies a resource to another object.SECTION 5.5 CASE STUDY: JAVA RMI 233

5.5 Case study: Java RMI

Java RMI extends the Java object model to provide support for distributed objects in the

Java language. In particular, it allows objects to invoke methods on remote objects using

the same syntax as for local invocations. In addition, type checking applies equally to

remote invocations as to local ones. However, an object making a remote invocation is

aware that its target is remote because it must handle RemoteExceptions; and the

implementor of a remote object is aware that it is remote because it must implement the

Remote interface. Although the distributed object model is integrated into Java in a

natural way, the semantics of parameter passing differ because the invoker and target are

remote from one another.

The programming of distributed applications in Java RMI should be relatively

simple because it is a single-language system – remote interfaces are defined in the Java

language. If a multiple-language system such as CORBA is used, the programmer needs

to learn an IDL and to understand how it maps onto the implementation language.

However, even in a single-language system, the programmer of a remote object must

consider its behaviour in a concurrent environment.

In the remainder of this introduction, we give an example of a remote interface,

then discuss the parameter-passing semantics with reference to the example. Finally, we

discuss the downloading of classes and the binder. The second section of this case study

discusses how to build client and server programs for the example interface. The third

section is concerned with the design and implementation of Java RMI. For full details

of Java RMI, see the tutorial on remote invocation [java.sun.com I].

In this case study, the CORBA case study in Chapter 8 and the discussion of web

services in Chapter 9, we use a shared whiteboard as an example. This is a distributed

program that allows a group of users to share a common view of a drawing surface

containing graphical objects, such as rectangles, lines and circles, each of which has

been drawn by one of the users. The server maintains the current state of a drawing by

providing an operation for clients to inform it about the latest shape one of their users

has drawn and keeping a record of all the shapes it has received. The server also provides

operations allowing clients to retrieve the latest shapes drawn by other users by polling

the server. The server has a version number (an integer) that it increments each time a

new shape arrives and attaches to the new shape. The server provides operations

allowing clients to enquire about its version number and the version number of each

shape, so that they may avoid fetching shapes that they already have.

Remote interfaces in Java RMI • Remote interfaces are defined by extending an

interface called Remote provided in the java.rmi package. The methods must throw

RemoteException, but application-specific exceptions may also be thrown. Figure 5.16

shows an example of two remote interfaces called Shape and ShapeList. In this example,

GraphicalObject is a class that holds the state of a graphical object – for example, its

type, its position, enclosing rectangle, line colour and fill colour – and provides

operations for accessing and updating its state. GraphicalObject must implement the

Serializable interface. Consider the interface Shape first: the getVersion method returns

an integer, whereas the getAllState method returns an instance of the class

GraphicalObject. Now consider the interface ShapeList: its newShape method passes an

instance of GraphicalObject as its argument but returns an object with a remote234 CHAPTER 5 REMOTE INVOCATION

interface (that is, a remote object) as its result. An important point to note is that both

ordinary objects and remote objects can appear as arguments and results in a remote

interface. The latter are always denoted by the name of their remote interface. In the next

subsection, we discuss how ordinary objects and remote objects are passed as arguments

and results.

Parameter and result passing • In Java RMI, the parameters of a method are assumed

to be input parameters and the result of a method is a single output parameter. Section

4.3.2 describes Java serialization, which is used for marshalling arguments and results

in Java RMI. Any object that is serializable – that is, that implements the Serializable

interface – can be passed as an argument or result in Java RMI. All primitive types and

remote objects are serializable. Classes for arguments and result values are downloaded

to the recipient by the RMI system where necessary.

Passing remote objects: When the type of a parameter or result value is defined as

a remote interface, the corresponding argument or result is always passed as a remote

object reference. For example, in Figure 5.16, line 2, the return value of the method

newShape is defined as Shape – a remote interface. When a remote object reference

is received, it can be used to make RMI calls on the remote object to which it refers.

Passing non-remote objects: All serializable non-remote objects are copied and

passed by value. For example, in Figure 5.16 (lines 2 and 1) the argument of

newShape and the return value of getAllState are both of type GraphicalObject,

which is serializable and is passed by value. When an object is passed by value, a new

object is created in the receiver’s process. The methods of this new object can be

invoked locally, possibly causing its state to differ from the state of the original object

in the sender’s process.

Thus, in our example, the client uses the method newShape to pass an instance of

GraphicalObject to the server; the server makes a remote object of type Shape

containing the state of the GraphicalObject and returns a remote object reference to it.

The arguments and return values in a remote invocation are serialized to a stream using

the method described in Section 4.3.2, with the following modifications:

Figure 5.16 Java Remote interfaces Shape and ShapeList

import java.rmi.\*;

import java.util.Vector;

public interface Shape extends Remote {

int getVersion() throws RemoteException;

GraphicalObject getAllState() throws RemoteException; 1

}

public interface ShapeList extends Remote {

Shape newShape(GraphicalObject g) throws RemoteException; 2

Vector allShapes() throws RemoteException;

int getVersion() throws RemoteException;

}SECTION 5.5 CASE STUDY: JAVA RMI 235

1. Whenever an object that implements the Remote interface is serialized, it is

replaced by its remote object reference, which contains the name of its (the remote

object’s) class.

2. When any object is serialized, its class information is annotated with the location

of the class (as a URL), enabling the class to be downloaded by the receiver.

Downloading of classes • Java is designed to allow classes to be downloaded from one

virtual machine to another. This is particularly relevant to distributed objects that

communicate by means of remote invocation. We have seen that non-remote objects are

passed by value and remote objects are passed by reference as arguments and results of

RMIs. If the recipient does not already possess the class of an object passed by value, its

code is downloaded automatically. Similarly, if the recipient of a remote object

reference does not already possess the class for a proxy, its code is downloaded

automatically. This has two advantages:

1. There is no need for every user to keep the same set of classes in their working

environment.

2. Both client and server programs can make transparent use of instances of new

classes whenever they are added.

As an example, consider the whiteboard program and suppose that its initial

implementation of GraphicalObject does not allow for text. A client with a textual

object can implement a subclass of GraphicalObject that deals with text and pass an

instance to the server as an argument of the newShape method. After that, other clients

may retrieve the instance using the getAllState method. The code of the new class will

be downloaded automatically from the first client to the server and then to other clients

as needed.

Figure 5.17 The Naming class of Java RMIregistry

void rebind (String name, Remote obj)

This method is used by a server to register the identifier of a remote object by name,

as shown in Figure 5.18, line 3.

void bind (String name, Remote obj)

This method can alternatively be used by a server to register a remote object by name,

but if the name is already bound to a remote object reference an exception is thrown.

void unbind (String name, Remote obj)

This method removes a binding.

Remote lookup(String name)

This method is used by clients to look up a remote object by name, as shown in Figure

5.20, line 1. A remote object reference is returned.

String [] list()

This method returns an array of Strings containing the names bound in the registry.236 CHAPTER 5 REMOTE INVOCATION

RMIregistry • The RMIregistry is the binder for Java RMI. An instance of RMIregistry

should normally run on every server computer that hosts remote objects. It maintains a

table mapping textual, URL-style names to references to remote objects hosted on that

computer. It is accessed by methods of the Naming class, whose methods take as an

argument a URL-formatted string of the form:

//computerName:port/objectName

where computerName and port refer to the location of the RMIregistry. If they are

omitted, the local computer and default port are assumed. Its interface offers the

methods shown in Figure 5.17, in which the exceptions are not listed – all of the methods

can throw a RemoteException.

Used in this way, clients must direct their lookup enquiries to particular hosts.

Alternatively, it is possible to set up a system-wide binding service. To achieve this, it

is necessary to run an instance of the RMIregistry in the networked environment and

then use the class LocateRegistry, which is in java.rmi.registry, to discover this registry.

More specifically, this class contains a getRegistry method that returns an object of type

Registry representing the remote binding service:

public static Registry getRegistry() throws RemoteException

Following this, it is then necessary to issue a call of rebind on this returned Registry

object to establish a connection with the remote RMIregistry.

5.5.1 Building client and server programs

This section outlines the steps necessary to produce client and server programs that use

the Remote interfaces Shape and ShapeList shown in Figure 5.16. The server program is

a simplified version of a whiteboard server that implements the two interfaces Shape and

ShapeList. We describe a simple polling client program and then introduce the callback

Figure 5.18 Java class ShapeListServer with main method

import java.rmi.\*;

import java.rmi.server.UnicastRemoteObject;

public class ShapeListServer{

public static void main(String args[]){

System.setSecurityManager(new RMISecurityManager());

try{

ShapeList aShapeList = new ShapeListServant(); 1

ShapeList stub = 2

(ShapeList) UnicastRemoteObject.exportObject(aShapeList,0);3

Naming.rebind("//bruno.ShapeList", stub ); 4

System.out.println("ShapeList server ready");

}catch(Exception e) {

System.out.println("ShapeList server main " + e.getMessage());}

}

}SECTION 5.5 CASE STUDY: JAVA RMI 237

technique that can be used to avoid the need to poll the server. Complete versions of the

classes illustrated in this section are available at www.cdk5.net/rmi.

Server program • The server is a whiteboard server: it represents each shape as a remote

object instantiated by a servant that implements the Shape interface and holds the state

of a graphical object as well as its version number; it represents its collection of shapes

by using another servant that implements the ShapeList interface and holds a collection

of shapes in a Vector.

The server program consists of a main method and a servant class to implement

each of its remote interfaces. The main method of the server class is shown in Figure

5.18, with the key steps contained in the lines marked 1 to 4:

• In line 1, the server creates an instance of ShapeListServant.

• Lines 2 and 3 use the method exportObject (defined on UnicastRemoteObject) to

make this object available to the RMI runtime, thereby making it available to

receive incoming invocations. The second parameter of exportObject specifies the

TCP port to be used for incoming invocations. It is normal practice to set this to

zero, implying that an anonymous port will be used (one that is generated by the

RMI runtime). Using UnicastRemoteObject ensures that the resultant object lives

only as long as the process in which it is created (an alternative is to make this an

Activatable object that is, one that lives beyond the server instance).

• Finally, line 4 binds the remote object to a name in the RMIregistry. Note that the

value bound to the name is a remote object reference, and its type is the type of its

remote interface – ShapeList.

The two servant classes are ShapeListServant, which implements the ShapeList

interface, and ShapeServant, which implements the Shape interface. Figure 5.19 gives

an outline of the class ShapeListServant.

Figure 5.19 Java class ShapeListServant implements interface ShapeList

import java.util.Vector;

public class ShapeListServant implements ShapeList {

private Vector theList; // contains the list of Shapes

private int version;

public ShapeListServant(){...}

public Shape newShape(GraphicalObject g) { 1

version++;

Shape s = new ShapeServant( g, version); 2

theList.addElement(s);

return s;

}

public Vector allShapes(){...}

public int getVersion() { ... }

}238 CHAPTER 5 REMOTE INVOCATION

The implementation of the methods of the remote interface in a servant class is

completely straightforward because it can be done without any concern for the details of

communication. Consider the method newShape in Figure 5.19 (line 1), which could be

called a factory method because it allows the client to request the creation of a servant.

It uses the constructor of ShapeServant, which creates a new servant containing the

GraphicalObject and version number passed as arguments. The type of the return value

of newShape is Shape – the interface implemented by the new servant. Before returning,

the method newShape adds the new shape to its vector that contains the list of shapes

(line 2).

The main method of a server needs to create a security manager to enable Java

security to apply the protection appropriate for an RMI server. A default security

manager called RMISecurityManager is provided. It protects the local resources to

ensure that the classes that are loaded from remote sites cannot have any effect on

resources such as files, but it differs from the standard Java security manager in allowing

the program to provide its own class loader and to use reflection. If an RMI server sets

no security manager, proxies and classes can only be loaded from the local classpath, in

order to protect the program from code that is downloaded as a result of remote method

invocations.

Client program • A simplified client for the ShapeList server is illustrated in Figure

5.20. Any client program needs to get started by using a binder to look up a remote

object reference. Our client sets a security manager and then looks up a remote object

reference for the remote object using the lookup operation of the RMIregistry (line 1).

Having obtained an initial remote object reference, the client continues by sending RMIs

to that remote object or to others discovered during its execution according to the needs

of its application. In our example, the client invokes the method allShapes in the remote

object (line 2) and receives a vector of remote object references to all of the shapes

currently stored in the server. If the client was implementing a whiteboard display, it

would use the server’s getAllState method in the Shape interface to retrieve each of the

graphical objects in the vector and display them in a window. Each time the user finishes

Figure 5.20 Java client of ShapeList

import java.rmi.\*;

import java.rmi.server.\*;

import java.util.Vector;

public class ShapeListClient{

public static void main(String args[]){

System.setSecurityManager(new RMISecurityManager());

ShapeList aShapeList = null;

try{

aShapeList = (ShapeList) Naming.lookup("//bruno.ShapeList"); 1

Vector sList = aShapeList.allShapes(); 2

} catch(RemoteException e) {System.out.println(e.getMessage());

}catch(Exception e) {System.out.println("Client: " + e.getMessage());}

}

}SECTION 5.5 CASE STUDY: JAVA RMI 239

drawing a graphical object, it will invoke the method newShape in the server, passing

the new graphical object as its argument. The client will keep a record of the latest

version number at the server, and from time to time it will invoke getVersion at the

server to find out whether any new shapes have been added by other users. If so, it will

retrieve and display them.

Callbacks • The general idea behind callbacks is that instead of clients polling the

server to find out whether some event has occurred, the server should inform its clients

whenever that event occurs. The term callback is used to refer to a server’s action of

notifying clients about an event. Callbacks can be implemented in RMI as follows:

• The client creates a remote object that implements an interface that contains a

method for the server to call. We refer to this as a callback object.

• The server provides an operation allowing interested clients to inform it of the

remote object references of their callback objects. It records these in a list.

• Whenever an event of interest occurs, the server calls the interested clients. For

example, the whiteboard server would call its clients whenever a graphical object

is added.

The use of callbacks avoids the need for a client to poll the objects of interest in the

server and its attendant disadvantages:

• The performance of the server may be degraded by the constant polling.

• Clients cannot notify users of updates in a timely manner.

However, callbacks have problems of their own. First, the server needs to have up-todate lists of the clients’ callback objects, but clients may not always inform the server

before they exit, leaving the server with incorrect lists. The leasing technique discussed

in Section 5.4.3 can be used to overcome this problem. The second problem associated

with callbacks is that the server needs to make a series of synchronous RMIs to the

callback objects in the list. See Chapter 6 for some ideas about solving the second

problem.

We illustrate the use of callbacks in the context of the whiteboard application. The

WhiteboardCallback interface could be defined as follows:

public interface WhiteboardCallback implements Remote {

void callback(int version) throws RemoteException;

};

This interface is implemented as a remote object by the client, enabling the server to

send the client a version number whenever a new object is added. But before the server

can do this, the client needs to inform the server about its callback object. To make this

possible, the ShapeList interface requires additional methods such as register and

deregister, defined as follows:

int register(WhiteboardCallback callback) throws RemoteException;

void deregister(int callbackId) throws RemoteException;

After the client has obtained a reference to the remote object with the ShapeList interface

(for example, in Figure 5.20, line 1) and created an instance of its callback object, it uses

the register method of ShapeList to inform the server that it is interested in receiving240 CHAPTER 5 REMOTE INVOCATION

callbacks. The register method returns an integer (the callbackId) referring to the

registration. When the client is finished it should call deregister to inform the server it

no longer requires callbacks. The server is responsible for keeping a list of interested

clients and notifying all of them each time its version number increases.

5.5.2 Design and implementation of Java RMI

The original Java RMI system used all of the components shown in Figure 5.15. But in

Java 1.2, the reflection facilities were used to make a generic dispatcher and to avoid the

need for skeletons. Prior to J2SE 5.0, the client proxies were generated by a compiler

called rmic from the compiled server classes (not from the definitions of the remote

interfaces). However, this step is no longer necessary with recent versions of J2SE,

which contain support for the dynamic generation of stub classes at runtime.

Use of reflection • Reflection is used to pass information in request messages about the

method to be invoked. This is achieved with the help of the class Method in the reflection

package. Each instance of Method represents the characteristics of a particular method,

including its class and the types of its arguments, return value and exceptions. The most

interesting feature of this class is that an instance of Method can be invoked on an object

of a suitable class by means of its invoke method. The invoke method requires two

arguments: the first specifies the object to receive the invocation and the second is an

array of Object containing the arguments. The result is returned as type Object.

To return to the use of the Method class in RMI: the proxy has to marshal

information about a method and its arguments into the request message. For the method

it marshals an object of class Method. It puts the arguments into an array of Objects and

then marshals that array. The dispatcher unmarshals the Method object and its

arguments in the array of Objects from the request message. As usual, the remote object

reference of the target will have been unmarshalled and the corresponding local object

reference obtained from the remote reference module. The dispatcher then calls the

Method object’s invoke method, supplying the target and the array of argument values.

When the method has been executed, the dispatcher marshals the result or any

exceptions into the reply message. Thus the dispatcher is generic – that is, the same

dispatcher can be used for all classes of remote object, and no skeletons are required.

Java classes supporting RMI • Figure 5.21 shows the inheritance structure of the classes

supporting Java RMI servers. The only class that the programmer need be aware of is

UnicastRemoteObject, which every simple servant class needs to extend. The class

UnicastRemoteObject extends an abstract class called RemoteServer, which provides

Figure 5.21 Classes supporting Java RMI

RemoteServer

UnicastRemoteObject

<servant class>

Activatable

RemoteObjectSECTION 5.6 SUMMARY 241

abstract versions of the methods required by remote servers. UnicastRemoteObject was

the first example of RemoteServer to be provided. Another called Activatable is

available for providing activatable objects. Further alternatives might provide for

replicated objects. The class RemoteServer is a subclass of RemoteObject that has an

instance variable holding the remote object reference and provides the following

methods:

equals This method compares remote object references.

toString: This method gives the contents of the remote object reference as a String.

readObject, writeObject: These methods deserialize/serialize remote objects.

In addition, the instanceOf operator can be used to test remote objects.

5.6 Summary

This chapter has discussed three paradigms for distributed programming – request-reply

protocols, remote procedure calls and remote method invocation. All of these paradigms

provide mechanisms for distributed independent entities (processes, objects,

components or services) to communicate directly with one another.

Request-reply protocols provide lightweight and minimal support for client-server

computing. Such protocols are often used in environments where overheads of

communication must be minimized – for example, in embedded systems. Their more

common role is to support either RPC or RMI, as discussed below.

The remote procedure call approach was a significant breakthrough in distributed

systems, providing higher-level support for programmers by extending the concept of a

procedure call to operate in a networked environment. This provides important levels of

transparency in distributed systems. However, due to their different failure and

performance characteristics and to the possibility of concurrent access to servers, it is

not necessarily a good idea to make remote procedure calls appear to be exactly the same

as local calls. Remote procedure calls provide a range of invocation semantics, from

maybe invocations through to at-most-once semantics.

The distributed object model is an extension of the local object model used in

object-based programming languages. Encapsulated objects form useful components in

a distributed system, since encapsulation makes them entirely responsible for managing

their own state, and local invocation of methods can be extended to remote invocation.

Each object in a distributed system has a remote object reference (a globally unique

identifier) and a remote interface that specifies which of its operations can be invoked

remotely.

Middleware implementations of RMI provide components (including proxies,

skeletons and dispatchers) that hide the details of marshalling, message passing and

locating remote objects from client and server programmers. These components can be

generated by an interface compiler. Java RMI extends local invocation to remote

invocation using the same syntax, but remote interfaces must be specified by extending

an interface called Remote and making each method throw a RemoteException. This

ensures that programmers know when they make remote invocations or implement

remote objects, enabling them to handle errors or to design objects suitable for

concurrent access.242 CHAPTER 5 REMOTE INVOCATION

EXERCISES

5.1 Define a class whose instances represent request and reply messages as illustrated in

Figure 5.4. The class should provide a pair of constructors, one for request messages and

the other for reply messages, showing how the request identifier is assigned. It should

also provide a method to marshal itself into an array of bytes and to unmarshal an array

of bytes into an instance. page 204

5.2 Program each of the three operations of the request-reply protocol in Figure 5.3, using

UDP communication, but without adding any fault-tolerance measures. You should use

the classes you defined in the previous chapter for remote object references (Exercise

4.13) and above for request and reply messages (Exercise 5.1). page 203

5.3 Give an outline of the server implementation, showing how the operations getRequest

and sendReply are used by a server that creates a new thread to execute each client

request. Indicate how the server will copy the requestId from the request message into

the reply message and how it will obtain the client IP address and port. page 203

5.4 For the operations of the request-reply protocols, provide the prototype of a new method

named sendRequest that has the request message, the Internet address of the receiver

machine, and its port as parameters. page 204

5.5 Describe a scenario in remote invocation in which request-reply protocols are required.

page 202

5.6 Describe the two components of message identifiers in request-reply communication.

page 205

5.7 Discuss whether the following operations are idempotent:

i) pressing a lift (elevator) request button;

ii) writing data to a file;

iii) appending data to a file.

Is it a necessary condition for idempotence that the operation should not be associated

with any state? page 206

5.8 Explain the mechanism of the request-reply protocol. How does it compensate for not

using the acknowledge message?

pages 207, 208

5.9 What are the three protocols used for implementing various types of request behaviour?

Which of these protocols can be used when the client requires no confirmation that the

operation has been executed? page 206

5.10 Why might the number of messages exchanged in a protocol be more significant to

performance than the total amount of data sent? Design a variant of the RRA protocol

in which the acknowledgement is piggy-backed on – that is, transmitted in the same

message as – the next request where appropriate, and otherwise sent as a separate

message. (Hint: use an extra timer in the client.) page 207EXERCISES 243

5.11 An Election interface provides two remote methods:

vote: This method has two parameters through which the client supplies the name

of a candidate (a string) and the ‘voter’s number’ (an integer used to ensure each

user votes once only). The voter’s numbers are allocated sparsely from the range

of integers to make them hard to guess.

result: This method has two parameters through which the server supplies the

client with the name of a candidate and the number of votes for that candidate.

Which of the parameters of these two procedures are input and which are output

parameters? page 211

5.12 Discuss the invocation semantics that can be achieved when the request-reply protocol

is implemented over a TCP/IP connection, which guarantees that data is delivered in the

order sent, without loss or duplication. Take into account all of the conditions causing a

connection to be broken. Section 4.2.4 and page 214

5.13 Define the interface to the Election service in CORBA IDL and Java RMI. Note that

CORBA IDL provides the type long for 32-bit integers. Compare the methods in the two

languages for specifying input and output arguments. Figure 5.8, Figure 5.16

5.14 The Election service must ensure that a vote is recorded whenever any user thinks they

have cast a vote.

Discuss the effect of maybe call semantics on the Election service.

Would at-least-once call semantics be acceptable for the Election service or would you

recommend at-most-once call semantics? page 215

5.15 A request-reply protocol is implemented over a communication service with omission

failures to provide at-least-once invocation semantics. In the first case the implementor

assumes an asynchronous distributed system. In the second case the implementor

assumes that the maximum time for the communication and the execution of a remote

method is T. In what way does the latter assumption simplify the implementation?

page 214

5.16 Outline an implementation for the Election service that ensures that its records remain

consistent when it is accessed concurrently by multiple clients. page 215

5.17 Assume the Election service is implemented in RMI and must ensure that all votes are

safely stored even when the server process crashes. Explain how this can be achieved

with reference to the implementation outline in your answer to Exercise 5.16.

pages 229–230

5.18 Show how to use Java reflection to construct the client proxy class for the Election

interface. Give the details of the implementation of one of the methods in this class,

which should call the method doOperation with the following signature:

byte[] doOperation (RemoteObjectRef o, Method m, byte[] arguments);

Hint: an instance variable of the proxy class should hold a remote object reference (see

Exercise 4.13). Figure 5.3, page 240244 CHAPTER 5 REMOTE INVOCATION

5.19 Show how to generate a client proxy class using a language such as C++ that does not

support reflection, for example from the CORBA interface definition given in your

answer to Exercise 5.13. Give the details of the implementation of one of the methods

in this class, which should call the method doOperation defined in Figure 5.3.

page 227

5.20 Explain how to use Java reflection to construct a generic dispatcher. Give Java code for

a dispatcher whose signature is:

public void dispatch(Object target, Method aMethod, byte[] args)

The arguments supply the target object, the method to be invoked and the arguments for

that method in an array of bytes. page 240

5.21 Exercise 5.18 required the client to convert Object arguments into an array of bytes

before invoking doOperation and Exercise 5.20 required the dispatcher to convert an

array of bytes into an array of Objects before invoking the method. Discuss the

implementation of a new version of doOperation with the following signature:

Object[] doOperation (RemoteObjectRef o, Method m, Object[] arguments);

which uses the ObjectOutputStream and ObjectInputStream classes to stream the

request and reply messages between client and server over a TCP connection. How

would these changes affect the design of the dispatcher? Section 4.3.2 and page 240

5.22 A client makes remote method invocations to a server. The client takes 5 milliseconds

to compute the arguments for each request, and the server takes 10 milliseconds to

process each request. The local operating system processing time for each send or

receive operation is 0.5 milliseconds, and the network time to transmit each request or

reply message is 3 milliseconds. Marshalling or unmarshalling takes 0.5 milliseconds

per message.

Calculate the time taken by the client to generate and return from two requests:

(i) if it is single-threaded;

(ii) if it has two threads that can make requests concurrently on a single

processor.

You can ignore context-switching times. Is there a need for asynchronous invocation if

the client and server processes are threaded? page 229

5.23 Design a remote object table that can support distributed garbage collection as well as

translating between local and remote object references. Give an example involving

several remote objects and proxies at various sites to illustrate the use of the table. Show

the changes in the table when an invocation causes a new proxy to be created. Then show

the changes in the table when one of the proxies becomes unreachable. page 231

5.24 A simpler version of the distributed garbage collection algorithm described in Section

5.4.3 just invokes addRef at the site where a remote object lives whenever a proxy is

created and removeRef whenever a proxy is deleted. Outline all the possible effects of

communication and process failures on the algorithm. Suggest how to overcome each of

these effects, but without using leases. page 231245

6

INDIRECT COMMUNICATION

6.1 Introduction

6.2 Group communication

6.3 Publish-subscribe systems

6.4 Message queues

6.5 Shared memory approaches

6.6 Summary

This chapter completes our tour of communication paradigms by examining indirect

communication; it builds on our studies of interprocess communication and remote

invocation in Chapters 4 and 5, respectively. The essence of indirect communication is to

communicate through an intermediary and hence have no direct coupling between the

sender and the one or more receivers. The important concepts of space and time

uncoupling are also introduced.

The chapter examines a range of indirect communication techniques:

• group communication, in which communication is via a group abstraction with the

sender unaware of the identity of the recipients;

• publish-subscribe systems, a family of approaches that all share the common

characteristic of disseminating events to multiple recipients through an

intermediary;

• message queue systems, wherein messages are directed to the familiar abstraction

of a queue with receivers extracting messages from such queues;

• shared memory–based approaches, including distributed shared memory and tuple

space approaches, which present an abstraction of a global shared memory to

programmers.

Case studies are used throughout the chapter to illustrate the main concepts introduced.246 CHAPTER 6 INDIRECT COMMUNICATION

6.1 Introduction

This chapter concludes our examination of communication paradigms by examining

indirect communication, building on the studies of interprocess communication and

remote invocation in Chapters 4 and 5, respectively. Indirection is a fundamental

concept in computer science, and its ubiquity and importance are captured nicely by the

following quote, which emerged from the Titan Project at the University of Cambridge

and is attributable to Roger Needham, Maurice Wilkes and David Wheeler:

All problems in computer science can be solved by another level of indirection.

In terms of distributed systems, the concept of indirection is increasingly applied to

communication paradigms.

Indirect communication is defined as communication between entities in a

distributed system through an intermediary with no direct coupling between the sender

and the receiver(s). The precise nature of the intermediary varies from approach to

approach, as will be seen in the rest of this chapter. In addition, the precise nature of

coupling varies significantly between systems, and again this will be brought out in the

text that follows. Note the optional plural associated with the receiver; this signifies that

many indirect communication paradigms explicitly support one-to-many communication.

The techniques considered in Chapters 4 and 5 are all based on a direct coupling

between a sender and a receiver, and this leads to a certain amount of rigidity in the

system in terms of dealing with change. To illustrate this, consider a simple client-server

interaction. Because of the direct coupling, it is more difficult to replace a server with

an alternative one offering equivalent functionality. Similarly, if the server fails, this

directly affects the client, which must explicitly deal with the failure. In contrast,

indirect communication avoids this direct coupling and hence inherits interesting

properties. The literature refers to two key properties stemming from the use of an

intermediary:

Space uncoupling, in which the sender does not know or need to know the identity

of the receiver(s), and vice versa. Because of this space uncoupling, the system

developer has many degrees of freedom in dealing with change: participants (senders

or receivers) can be replaced, updated, replicated or migrated.

Time uncoupling, in which the sender and receiver(s) can have independent

lifetimes. In other words, the sender and receiver(s) do not need to exist at the same

time to communicate. This has important benefits, for example, in more volatile

environments where senders and receivers may come and go.

For these reasons, indirect communication is often used in distributed systems where

change is anticipated – for example, in mobile environments where users may rapidly

connect to and disconnect from the global network – and must be managed to provide

more dependable services. Indirect communication is also heavily used for event

dissemination in distributed systems where the receivers may be unknown and liable to

change – for example, in managing event feeds in financial systems, as featured in

Chapter 1. Indirect communication is also exploited in key parts of the Google

infrastructure, as discussed in the major case study in Chapter 21.SECTION 6.1 INTRODUCTION 247

The discussion above charts the advantages associated with indirect

communication. The main disadvantage is that there will inevitably be a performance

overhead introduced by the added level of indirection. Indeed, the quote above on

indirection is often paired by the following quote, attributable to Jim Gray:

There is no performance problem that cannot be solved by eliminating a

level of indirection.

In addition, systems developed using indirect communication can be more difficult to

manage precisely because of the lack of any direct (space or time) coupling.

A closer look at space and time uncoupling • It may be assumed that indirection implies

both space and time uncoupling, but this is not always the case. The precise relationship

is summarized in Figure 6.1.

From this table, it is clear that most of the techniques considered in this book are

either coupled in both time and space or indeed uncoupled in both dimensions. The topleft box represents the communication paradigms featured in Chapters 4 and 5 where

communication is direct with no space or time uncoupling. For example, message

passing is both directed towards a particular entity and requires the receiver to be present

at the time of the message send (but see Exercise 6.2 for an added dimension introduced

by DNS name resolution). The range of remote invocation paradigms are also coupled

in both space and time. The bottom-right box represents the main indirect

communication paradigms that exhibit both properties. A small number of

communication paradigms sit outside these two areas:

• IP multicast, as featured in Chapter 4, is space-uncoupled but time-coupled. It is

space-uncoupled because messages are directed towards the multicast group, not

any particular receiver. It is time-coupled, though, as all receivers must exist at the

time of the message send to receive the multicast. Some implementations of group

communication and indeed publish-subscribe systems, also fall into this category

(see Section 6.6). This example illustrates the importance of persistency in the

Figure 6.1 Space and time coupling in distributed systems

Time-coupled Time-uncoupled

Space coupling

Properties: Communication directed

towards a given receiver or receivers;

receiver(s) must exist at that moment in

time

Examples: Message passing, remote

invocation (see Chapters 4 and 5)

Properties: Communication directed

towards a given receiver or receivers;

sender(s) and receiver(s) can have

independent lifetimes

Examples: See Exercise 6.3

Space uncoupling

Properties: Sender does not need to

know the identity of the receiver(s);

receiver(s) must exist at that moment in

time

Examples: IP multicast (see Chapter 4)

Properties: Sender does not need to know

the identity of the receiver(s); sender(s)

and receiver(s) can have independent

lifetimes

Examples: Most indirect communication

paradigms covered in this chapter248 CHAPTER 6 INDIRECT COMMUNICATION

communication channel to achieve time uncoupling – that is, the communication

paradigm must store messages so that they can be delivered when the receiver(s)

is ready to receive. IP multicast does not support this level of persistency.

• The case in which communication is space-coupled but time-uncoupled is more

subtle. Space coupling implies that the sender knows the identity of a specific

receiver or receivers, but time uncoupling implies that the receiver or receivers

need not exist at the time of sending. Exercises 6.3 and 6.4 invite the reader to

consider whether such a paradigm exists or could be constructed.

Returning to our definition, we treat all paradigms that involve an intermediary as

indirect and recognize that the precise level of coupling will vary from system to system.

We revisit the properties of different indirect communication paradigms in Section 6.6,

once we have had a chance to study the precise characteristics of each approach.

The relationship with asynchronous communication • Note that, to fully understand this

area, it is important to distinguish between asynchronous communication (as defined in

Chapter 4) and time uncoupling. In asynchronous communication, a sender sends a

message and then continues (without blocking), and hence there is no need to meet in

time with the receiver to communicate. Time uncoupling adds the extra dimension that

the sender and receiver(s) can have independent existences; for example, the receiver

may not exist at the time communication is initiated. Eugster et al. also recognize the

important distinction between asynchronous communication (synchronization

uncoupling) and time uncoupling [2003].

Many of the techniques examined in this chapter are time-uncoupled and

asynchronous, but a few, such as the MessageDispatcher and RpcDispatcher operations

in JGroups, discussed in Section 6.2.3, offer a synchronous service over indirect

communication.

The rest of the chapter examines specific examples of indirect communication

starting with group communication in Section 6.2. Section 6.3 then examines the

fundamentals of publish-subscribe systems, with Section 6.4 examining the contrasting

approach offered by message queues. Following this, Section 6.5 considers approaches

based on shared memory abstractions, specifically distributed shared memory and tuple

space–based approaches.

6.2 Group communication

Group communication provides our first example of an indirect communication

paradigm. Group communication offers a service whereby a message is sent to a group

and then this message is delivered to all members of the group. In this action, the sender

is not aware of the identities of the receivers. Group communication represents an

abstraction over multicast communication and may be implemented over IP multicast or

an equivalent overlay network, adding significant extra value in terms of managing

group membership, detecting failures and providing reliability and ordering guarantees.

With the added guarantees, group communication is to IP multicast what TCP is to the

point-to-point service in IP.SECTION 6.2 GROUP COMMUNICATION 249

Group communication is an important building block for distributed systems, and

particularly reliable distributed systems, with key areas of application including:

• the reliable dissemination of information to potentially large numbers of clients,

including in the financial industry, where institutions require accurate and up-todate access to a wide variety of information sources;

• support for collaborative applications, where again events must be disseminated

to multiple users to preserve a common user view – for example, in multiuser

games, as discussed in Chapter 1;

• support for a range of fault-tolerance strategies, including the consistent update of

replicated data (as discussed in detail in Chapter 18) or the implementation of

highly available (replicated) servers;

• support for system monitoring and management, including for example load

balancing strategies.

We look at group communication in more detail below, examining the programming

model offered and the associated implementation issues. We examine the JGroups

toolkit as a case study of a group communication service.

6.2.1 The programming model

In group communication, the central concept is that of a group with associated group

membership, whereby processes may join or leave the group. Processes can then send a

message to this group and have it propagated to all members of the group with certain

guarantees in terms of reliability and ordering. Thus, group communication implements

multicast communication, in which a message is sent to all the members of the group by

a single operation. Communication to all processes in the system, as opposed to a

subgroup of them, is known as broadcast, whereas communication to a single process

is known as unicast.

The essential feature of group communication is that a process issues only one

multicast operation to send a message to each of a group of processes (in Java this

operation is aGroup.send(aMessage)) instead of issuing multiple send operations to

individual processes.

The use of a single multicast operation instead of multiple send operations

amounts to much more than a convenience for the programmer: it enables the

implementation to be efficient in its utilization of bandwidth. It can take steps to send

the message no more than once over any communication link, by sending it over a

distribution tree; and it can use network hardware support for multicast where this is

available. The implementation can also minimize the total time taken to deliver the

message to all destinations, as compared with transmitting it separately and serially.

To see these advantages, compare the bandwidth utilization and the total

transmission time taken when sending the same message from a computer in London to

two computers on the same Ethernet in Palo Alto, (a) by two separate UDP sends, and (b)

by a single IP multicast operation. In the former case, two copies of the message are sent

independently, and the second is delayed by the first. In the latter case, a set of multicastaware routers forward a single copy of the message from London to a router on the

destination LAN in California. That router then uses hardware multicast (provided by the

Ethernet) to deliver the message to both destinations at once, instead of sending it twice.250 CHAPTER 6 INDIRECT COMMUNICATION

The use of a single multicast operation is also important in terms of delivery

guarantees. If a process issues multiple independent send operations to individual

processes, then there is no way for the implementation to provide guarantees that affect

the group of processes as a whole. If the sender fails halfway through sending, then some

members of the group may receive the message while others do not. In addition, the

relative ordering of two messages delivered to any two group members is undefined.

Group communication, however, has the potential to offer a range of guarantees in terms

of reliability and ordering, as discussed in Section 6.2.2 below.

Group communication has been the subject of many research projects, including

the V-system [Cheriton and Zwaenepoel 1985], Chorus [Rozier et al. 1988], Amoeba

[Kaashoek et al. 1989; Kaashoek and Tanenbaum 1991], Trans/Total [Melliar-Smith et

al. 1990], Delta-4 [Powell 1991], Isis [Birman 1993], Horus [van Renesse et al. 1996],

Totem [Moser et al. 1996] and Transis [Dolev and Malki 1996] – and we shall cite other

notable work in the course of this chapter and indeed throughout the book (particularly

in Chapters 15 and 18).

Process groups and object groups • Most work on group services focuses on the

concept of process groups, that is, groups where the communicating entities are

processes. Such services are relatively low-level in that:

• Messages are delivered to processes and no further support for dispatching is

provided.

• Messages are typically unstructured byte arrays with no support for marshalling

of complex data types (as provided, for example, in RPC or RMI – see Chapter 5).

The level of service provided by process groups is therefore similar to that of sockets,

as discussed in Chapter 4. In contrast, object groups provide a higher-level approach to

group computing. An object group is a collection of objects (normally instances of the

same class) that process the same set of invocations concurrently, with each returning

responses. Client objects need not be aware of the replication. They invoke operations

on a single, local object, which acts as a proxy for the group. The proxy uses a group

communication system to send the invocations to the members of the object group.

Object parameters and results are marshalled as in RMI and the associated calls are

dispatched automatically to the right destination objects/methods.

Electra [Maffeis 1995] is a CORBA-compliant system that supports object

groups. An Electra group can be interfaced to any CORBA-compliant application.

Electra was originally built on top of the Horus group communication system, which it

uses to manage the membership of the group and to multicast invocations. In

‘transparent mode’, the local proxy returns the first available response to a client object.

In ‘non-transparent mode’, the client object can access all the responses returned by the

group members. Electra uses an extension of the standard CORBA Object Request

Broker interface, with functions for creating and destroying object groups and managing

their membership. Eternal [Moser et al. 1998] and the Object Group Service [Guerraoui

et al. 1998] also provide CORBA-compliant support for object groups.

Despite the promise of object groups, however, process groups still dominate in

terms of usage. For example, the popular JGroups toolkit, discussed in Section 6.2.3, is

a classical process group approach.SECTION 6.2 GROUP COMMUNICATION 251

Other key distinctions • A wide range of group communication services has been

developed, and they vary in the assumptions they make:

Closed and open groups: A group is said to be closed if only members of the group

may multicast to it (Figure 6.2). A process in a closed group delivers to itself any

message that it multicasts to the group. A group is open if processes outside the group

may send to it. (The categories ‘open’ and ‘closed’ also apply with analogous

meanings to mailing lists.) Closed groups of processes are useful, for example, for

cooperating servers to send messages to one another that only they should receive.

Open groups are useful, for example, for delivering events to groups of interested

processes.

Overlapping and non-overlapping groups: In overlapping groups, entities (processes or objects) may be members of multiple groups, and non-overlapping groups imply that membership does not overlap (that is, any process belongs to at most one

group). Note that in real-life systems, it is realistic to expect that group membership

will overlap.

Synchronous and asynchronous systems: There is a requirement to consider group

communication in both environments.

Such distinctions can have a significant impact on the underlying multicast algorithms.

For example, some algorithms assume that groups are closed. The same effect as

openness can be achieved with a closed group by picking a member of the group and

sending it a message (one-to-one) for it to multicast to its group. Rodrigues et al. [1998]

discuss multicast to open groups. Issues related to open and closed groups arise in

Chapter 15, when algorithms for reliability and ordering are considered. That chapter

also considers the impact of overlapping groups and whether the system is synchronous

or asynchronous on such protocols.

Figure 6.2 Open and closed groups

Closed group Open group252 CHAPTER 6 INDIRECT COMMUNICATION

6.2.2 Implementation issues

We now turn to implementation issues for group communication services, discussing the

properties of the underlying multicast service in terms of reliability and ordering and

also the key role of group membership management in dynamic environments, where

processes can join and leave or fail at any time.

Reliability and ordering in multicast • In group communication, all members of a group

must receive copies of the messages sent to the group, generally with delivery

guarantees. The guarantees include agreement on the set of messages that every process

in the group should receive and on the delivery ordering across the group members.

Group communication systems are extremely sophisticated. Even IP multicast,

which provides minimal delivery guarantees, requires a major engineering effort.

So far, we have discussed reliability and ordering in rather general terms. We now

look in more detail at what such properties mean.

Reliability in one-to-one communication was defined in Section 2.4.2 in terms of

two properties: integrity (the message received is the same as the one sent, and no

messages are delivered twice) and validity (any outgoing message is eventually

delivered). The interpretation for reliable multicast builds on these properties, with

integrity defined the same way in terms of delivering the message correctly at most once,

and validity guaranteeing that a message sent will eventually be delivered. To extend the

semantics to cover delivery to multiple receivers, a third property is added – that of

agreement, stating that if the message is delivered to one process, then it is delivered to

all processes in the group.

As well as reliability guarantees, group communication demands extra guarantees

in terms of the relative ordering of messages delivered to multiple destinations. Ordering

is not guaranteed by underlying interprocess communication primitives. For example, if

multicast is implemented by a series of one-to-one messages, they may be subject to

arbitrary delays. Similar problems may occur if using IP multicast. To counter this,

group communication services offer ordered multicast, with the option of one or more

of the following properties (with hybrid solutions also possible):

FIFO ordering: First-in-first-out (FIFO) ordering (also referred to as source

ordering) is concerned with preserving the order from the perspective of a sender

process, in that if a process sends one message before another, it will be delivered in

this order at all processes in the group.

Causal ordering: Causal ordering takes into account causal relationships between

messages, in that if a message happens before another message in the distributed

system this so-called causal relationship will be preserved in the delivery of the

associated messages at all processes (see Chapter 14 for a detailed discussion of the

meaning of ‘happens before’).

Total ordering: In total ordering, if a message is delivered before another message

at one process, then the same order will be preserved at all processes.

Reliability and ordering are examples of coordination and agreement in distributed

systems, and hence further consideration of this is deferred to Chapter 15, which focuses

exclusively on this topic. In particular, Chapter 15 provides more complete definitionsSECTION 6.2 GROUP COMMUNICATION 253

of integrity, validity, agreement and the various ordering properties and also examines

in detail algorithms to realize reliable and ordered multicast.

Group membership management • The key elements of group communication

management are summarized in Figure 6.3, which shows an open group. This diagram

illustrates the important role of group membership management in maintaining an

accurate view of the current membership, given that entities may join, leave or indeed

fail. In more detail, a group membership service has four main tasks:

Providing an interface for group membership changes: The membership service

provides operations to create and destroy process groups and to add or withdraw a

process to or from a group. In most systems, a single process may belong to several

groups at the same time (overlapping groups, as defined above). This is true of IP

multicast, for example.

Failure detection: The service monitors the group members not only in case they

should crash, but also in case they should become unreachable because of a

communication failure. The detector marks processes as Suspected or Unsuspected.

The service uses the failure detector to reach a decision about the group’s

membership: it excludes a process from membership if it is suspected to have failed

or to have become unreachable.

Notifying members of group membership changes: The service notifies the group’s

members when a process is added, or when a process is excluded (through failure or

when the process is deliberately withdrawn from the group).

Performing group address expansion: When a process multicasts a message, it

supplies the group identifier rather than a list of processes in the group. The

membership management service expands the identifier into the current group

Figure 6.3 The role of group membership management

Join

Group

address

expansion

Multicast

communication

Group

send

Fail Group membership

management

Leave

Group254 CHAPTER 6 INDIRECT COMMUNICATION

membership for delivery. The service can coordinate multicast delivery with

membership changes by controlling address expansion. That is, it can decide

consistently where to deliver any given message, even though the membership may

be changing during delivery.

Note that IP multicast is a weak case of a group membership service, with some but not

all of these properties. It does allow processes to join or leave groups dynamically and

it performs address expansion, so that senders need only provide a single IP multicast

address as the destination for a multicast message. But IP multicast does not itself

provide group members with information about current membership, and multicast

delivery is not coordinated with membership changes. Achieving these properties is

complex and requires what is known as view-synchronous group communication.

Further consideration of this important issue is deferred to Chapter 18, which discusses

the maintenance of group views and how to realize view-synchronous group

communication in the context of supporting replication in distributed systems.

In general, the need to maintain group membership has a significant impact on the

utility of group-based approaches. In particular, group communication is most effective

in small-scale and static systems and does not operate as well in larger-scale

environments or environments with a high degree of volatility. This can be traced to the

need for a form of synchrony assumption. Ganesh et al [2003] present a more

probabilistic approach to group membership designed to operate in more large-scale and

dynamic environments, using an underlying gossip protocol (see Section 10.5.3).

Researchers have also developed group membership protocols specifically for ad hoc

networks and mobile environments [Prakash and Baldoni 1998; Roman et al. 2001; Liu

et al. 2005].

6.2.3 Case study: the JGroups toolkit

JGroups is a toolkit for reliable group communication written in Java. The toolkit is a

part of the lineage of group communication tools that have emerged from Cornell

University, building on the fundamental concepts developed in ISIS [Birman 1993],

Horus [van Renesse et al. 1996] and Ensemble [van Renesse et al. 1998]. The toolkit is

now maintained and developed by the JGroups open source community

[www.jgroups.org], which is part of the JBoss middleware community, as discussed in

Chapter 8 [www.jboss.org].

JGroups supports process groups in which processes are able to join or leave a

group, send a message to all members of the group or indeed to a single member, and

receive messages from the group. The toolkit supports a variety of reliability and

ordering guarantees, which are discussed in more detail below, and also offers a group

membership service.

The architecture of JGroups is shown in Figure 6.4, which shows the main

components of the JGroups implementation:

• Channels represent the most primitive interface for application developers,

offering the core functions of joining, leaving, sending and receiving.

• Building blocks offer higher-level abstractions, building on the underlying service

offered by channels.SECTION 6.2 GROUP COMMUNICATION 255

• The protocol stack provides the underlying communication protocol, constructed

as a stack of composable protocol layers.

We look at each in turn below.

Channels • A process interacts with a group through a channel object, which acts as a

handle onto a group. When created, it is disconnected, but a subsequent connect

operation binds that handle to a particular named group; if the named group does not

exist, it is implicitly created at the time of the first connect. To leave the group, the

process executes the corresponding disconnect operation. A close operation is also

provided to render the channel unusable. Note that a channel can only be connected to

one group at a time; if a process wants to connect to two or more groups, it must create

multiple channels. When connected, a process can send or receive via a channel.

Messages are sent by reliable multicast, with the precise semantics defined by the

protocol stack deployed (as discussed further below).

A range of other operations are defined on channels, most notably to return

management information associated with the channel. For example, getView returns the

current view defined in terms of the current member list, while getState returns the

historical application state associated with the group (this can be used, for example, by

a new group member to catch up with previous events).

Note that the term channel should not be confused with channel-based publishsubscribe, as introduced in Section 6.3.1. A channel in JGroups is synonymous with an

instance of a group as defined in Section 6.2.1.

Figure 6.4 The architecture of JGroups

Network

UDP

FRAG

MERGE

GMS

CAUSAL

Protocol stack

Channel

Building

blocks

Applications256 CHAPTER 6 INDIRECT COMMUNICATION

We illustrate the use of channels further by a simple example, a service whereby

an intelligent fire alarm can send a “Fire!” multicast message to any registered receivers.

The code for the fire alarm is as shown in Figure 6.5.

When an alarm is raised, the first step is to create a new instance of JChannel (the

class representing channels in JGroups) and then connect to a group called

AlarmChannel. If this is the first connect, then the group will be created at this stage

(unlikely in this example, or the alarm is not going to be very effective). The constructor

for a message takes three parameters: the destination, the source and the payload. In this

case, the destination is null, which specifies that the message is to be sent to all members

(if an address is specified, it is sent to that address only). The source is also null; this

need not be provided in JGroups as it will be included automatically. The payload is an

unstructured byte array that is delivered to all members of the group through the send

method. The code to create a new instance of the FireAlarmJG class and then raise an

alarm would be:

FireAlarmJG alarm = new FireAlarmJG();

alarm.raise();

The corresponding code for the receiver end has a similar structure and is shown in

Figure 6.6. In this case, however, after connecting a receive method is called. This

method takes one parameter, a timeout. If it is set to zero, as in this case, the receive

message will block until a message is received. Note that in JGroups incoming messages

are buffered and receive returns the top element in the buffer. If no messages are present,

then receive blocks awaiting the next message. Strictly speaking, receive can return a

range of object types – for example, notification of a change in membership or of a

suspected failure of a group member (hence the cast to Message above).

A given receiver must include the following code to await an alarm:

FireAlarmConsumerJG alarmCall = new FireAlarmConsumerJG();

String msg = alarmCall.await();

System.out.println("Alarm received: " + msg);

Figure 6.5 Java class FireAlarmJG

import org.jgroups.JChannel;

public class FireAlarmJG {

public void raise() {

try {

JChannel channel = new JChannel();

channel.connect("AlarmChannel");

Message msg = new Message(null, null, "Fire!");

channel.send(msg);

}

catch(Exception e) {

}

}

}SECTION 6.2 GROUP COMMUNICATION 257

Building blocks • Building blocks are higher-level abstractions on top of the channel

class discussed above. Channels are similar in level to sockets. Building blocks are

analogous to the more advanced communication paradigms discussed in Chapter 5,

offering support for commonly occurring patterns of communication (but in this case

targeted at multicast communication). Examples of building blocks in JGroups are:

• MessageDispatcher is the most intuitive of the building blocks offered in

JGroups. In group communication, it is often useful for a sender to send a message

to a group and then wait for some or all of the replies. MessageDispatcher

supports this by providing a castMessage method that sends a message to a group

and blocks until a specified number of replies are received (for example, until a

specified number n, a majority, or all messages are received).

• RpcDispatcher takes a specific method (together with optional parameters and

results) and then invokes this method on all objects associated with a group. As

with MessageDispatcher, the caller can block awaiting some or all of the replies.

• NotificationBus is an implementation of a distributed event bus, in which an event

is any serializable Java object. This class is often used to implement consistency

in replicated caches.

The protocol stack • JGroups follows the architectures offered by Horus and Ensemble

by constructing protocol stacks out of protocol layers (initially referred to as microprotocols in the literature [van Renesse et al. 1996, 1998]). In this approach, a protocol

is a bidirectional stack of protocol layers with each layer implementing the following

two methods:

public Object up (Event evt);

public Object down (Event evt);

Protocol processing therefore happens by passing events up and down the stack. In

JGroups, events may be incoming or outgoing messages or management events, for

example related to view changes. Each layer can carry out arbitrary processing on the

Figure 6.6 Java class FireAlarmConsumerJG

import org.jgroups.JChannel;

public class FireAlarmConsumerJG {

public String await() {

try {

JChannel channel = new JChannel();

channel.connect("AlarmChannel");

Message msg = (Message) channel.receive(0);

return (String) msg.GetObject();

}

catch(Exception e) {

return null;

}

}

}258 CHAPTER 6 INDIRECT COMMUNICATION

message, including modifying its contents, adding a header or indeed dropping or reordering the message.

To illustrate the concept further, let us examine the protocol stack shown in Figure

6.4. This shows a protocol that consists of five layers:

• The layer referred to as UDP is the most common transport layer in JGroups. Note

that, despite the name, this is not entirely equivalent to the UDP protocol; rather,

the layer utilizes IP multicast for sending to all members in a group and UDP

datagrams specifically for point-to-point communication. This layer therefore

assumes that IP multicast is available. If it is not, the layer can be configured to

send a series of unicast messages to members, relying on another layer for

membership discovery (in particular, a layer known as PING). For larger-scale

systems operating over wide area networks, a TCP layer may be preferred (using

the TCP protocol to send unicast messages and again relying on PING for

membership discovery).

• FRAG implements message packetization and is configurable in terms of the

maximum message size (8,192 bytes by default).

• MERGE is a protocol that deals with unexpected network partitioning and the

subsequent merging of subgroups after the partition. A series of alternative merge

layers are actually available, ranging from the simple to ones that deal with, for

example, state transfer.

• GMS implements a group membership protocol to maintain consistent views of

membership across the group (see Chapter 18 for further details of algorithms for

group membership management).

• CAUSAL implements causal ordering, introduced in Section 6.2.2 (and discussed

further in Chapter 15).

A wide range of other protocol layers are available, including protocols for FIFO and

total ordering, for membership discovery and failure detection, for encryption of

messages and for implementing flow-control strategies (see the JGroups web site for

details [www.jgroups.org]). Note that because all layers implement the same interface,

they can be combined in any order, although many of the resultant protocol stacks would

not make sense. All members of a group must share the same protocol stack.

6.3 Publish-subscribe systems

We now turn our attention to the area of publish-subscribe systems [Baldoni and

Virgillito 2005], sometimes also referred to as distributed event-based systems [Muhl et

al. 2006]. These are the most widely used of all the indirect communication techniques

discussed in this chapter. Chapter 1 has already highlighted that many classes of system

are fundamentally concerned with the communication and processing of events (for

example financial trading systems). More specifically, whereas many systems naturally

map onto a request-reply or a remote invocation pattern of interaction as discussed in

Chapter 5, many do not and are more naturally modelled by the more decoupled and

reactive style of programming offered by events.SECTION 6.3 PUBLISH-SUBSCRIBE SYSTEMS 259

A publish-subscribe system is a system where publishers publish structured

events to an event service and subscribers express interest in particular events through

subscriptions which can be arbitrary patterns over the structured events. For example, a

subscriber could express an interest in all events related to this textbook, such as the

availability of a new edition or updates to the related web site. The task of the publishsubscribe system is to match subscriptions against published events and ensure the

correct delivery of event notifications. A given event will be delivered to potentially

many subscribers, and hence publish-subscribe is fundamentally a one-to-many

communications paradigm.

Applications of publish-subscribe systems • Publish-subscribe systems are used in a

wide variety of application domains, particularly those related to the large-scale

dissemination of events. Examples include:

• financial information systems;

• other areas with live feeds of real-time data (including RSS feeds);

• support for cooperative working, where a number of participants need to be

informed of events of shared interest;

• support for ubiquitous computing, including the management of events emanating

from the ubiquitous infrastructure (for example, location events);

• a broad set of monitoring applications, including network monitoring in the

Internet.

Publish-subscribe is also a key component of Google’s infrastructure, including for

example the dissemination of events related to advertisements, such as ‘ad clicks’, to

interested parties (see Chapter 21).

To illustrate the concept further, we consider a simple dealing room system as an

example of the broader class of financial information systems.

Dealing room system: Consider a simple dealing room system whose task is to allow

dealers using computers to see the latest information about the market prices of the

stocks they deal in. The market price for a single named stock is represented by an

associated object. The information arrives in the dealing room from several different

external sources in the form of updates to some or all of the objects representing the

stocks and is collected by processes we call information providers. Dealers are typically

interested only in their own specialist stocks. A dealing room system could be

implemented by processes with two different tasks:

• An information provider process continuously receives new trading information

from a single external source. Each of the updates is regarded as an event. The

information provider publishes such events to the publish-subscribe system for

delivery to all of the dealers who have expressed an interest in the corresponding

stock. There will be a separate information provider process for each external

source.

• A dealer process creates a subscription representing each named stock that the

user asks to have displayed. Each subscription expresses an interest in events

related to a given stock at the relevant information provider. It then receives all the

information sent to it in notifications and displays it to the user. The

communication of notifications is illustrated in Figure 6.7.260 CHAPTER 6 INDIRECT COMMUNICATION

Characteristics of publish-subscribe systems • Publish-subscribe systems have two

main characteristics:

Heterogeneity: When event notifications are used as a means of communication,

components in a distributed system that were not designed to interoperate can be

made to work together. All that is required is that event-generating objects publish

the types of events they offer, and that other objects subscribe to patterns of events

and provide an interface for receiving and dealing with the resultant notifications. For

example, Bates et al. [1996] describe how publish-subscribe systems can be used to

connect heterogeneous components in the Internet. They describe a system in which

applications can be made aware of users’ locations and activities, such as using

computers, printers or electronically tagged books. They envisage its future use in the

context of a home network supporting commands such as: ‘if the children come

home, turn on the central heating’.

Asynchronicity: Notifications are sent asynchronously by event-generating

publishers to all the subscribers that have expressed an interest in them to prevent

publishers needing to synchronize with subscribers – publishers and subscribers need

to be decoupled. Mushroom [Kindberg et al. 1996] is an object-based publishsubscribe system designed to support collaborative work, in which the user interface

displays objects representing users and information objects such as documents and

notepads within shared workspaces called network places. The state of each place is

replicated at the computers of users currently in that place. Events are used to

describe changes to objects and to a user’s focus of interest. For example, an event

Figure 6.7 Dealing room system

Dealer's computer

Information

provider

Dealer

External

source

External

source

Information

provider

Dealer

Dealer Dealer

Notification

Notification

Notification

Notification

Notification Notification

Notification

Notification

Notification

Notification

Dealer's computer

Dealer's computer Dealer's computerSECTION 6.3 PUBLISH-SUBSCRIBE SYSTEMS 261

could specify that a particular user has entered or left a place or has performed a

particular action on an object. Each replica of any object to which particular types of

events are relevant expresses an interest in them through a subscription and receives

notifications when they occur. But subscribers to events are decoupled from objects

experiencing events, because different users are active at different times.

In addition, a variety of different delivery guarantees can be provided for notifications

– the one that is chosen should depend on the application requirements. For example, if

IP multicast is used to send notifications to a group of receivers, the failure model will

relate to the one described for IP multicast in Section 4.4.1 and will not guarantee that

any particular recipient will receive a particular notification message. This is adequate

for some applications – for example, to deliver the latest state of a player in an Internet

game – because the next update is likely to get through.

However, other applications have stronger requirements. Consider the dealing

room application: to be fair to the dealers interested in a particular stock, we require that

all the dealers for the same stock receive the same information. This implies that a

reliable multicast protocol should be used.

In the Mushroom system mentioned above, notifications about changes in object

state are delivered reliably to a server, whose responsibility it is to maintain up-to-date

copies of objects. However, notifications may also be sent to object replicas in users’

computers by means of unreliable multicast; in the case that the latter lose notifications,

they can retrieve an object’s state from the server. When the application requires it,

notifications may be ordered and sent reliably to object replicas.

Some applications have real-time requirements.These include events in a nuclear

power station or a hospital patient monitor. It is possible to design multicast protocols

that provide real-time guarantees as well as reliability and ordering in a system that

satisfies the properties of a synchronous distributed system.

We discuss publish-subscribe systems in more detail in the following sections,

considering the programming model they offer before examining some of the key

implementation challenges, particularly related to large-scale dissemination of events in

the Internet.

6.3.1 The programming model

The programming model in publish-subscribe systems is based on a small set of

operations, captured in Figure 6.8. Publishers disseminate an event e through a

publish(e) operation and subscribers express an interest in a set of events through

subscriptions. In particular, they achieve this through a subscribe(f) operation where f

refers to a filter – that is, a pattern defined over the set of all possible events. The

expressiveness of filters (and hence of subscriptions) is determined by the subscription

model; which we discuss in more detail below. Subscribers can later revoke this interest

through a corresponding unsubscribe(f) operation. When events arrive at a subscriber,

the events are delivered using a notify(e) operation.

Some systems complement the above set of operations by introducing the concept

of advertisements. With advertisements, publishers have the option of declaring the

nature of future events through an advertise(f) operation. The advertisements are defined

in terms of the types of events of interest (these happen to take the same form as filters).262 CHAPTER 6 INDIRECT COMMUNICATION

In other words, subscribers declare their interests in terms of subscriptions and

publishers optionally declare the styles of events they will generate through

advertisements. Advertisements can be revoked through a call of unadvertise(f).

As mentioned above, the expressiveness of publish-subscribe systems is

determined by the subscription (filter) model, with a number of schemes defined and

considered here in increasing order of sophistication:

Channel-based: In this approach, publishers publish events to named channels and

subscribers then subscribe to one of these named channels to receive all events sent

to that channel. This is a rather primitive scheme and the only one that defines a

physical channel; all other schemes employ some form of filtering over the content

of events as we shall see below. Although simple, this scheme has been used

successfully in the CORBA Event Service (see Chapter 8).

Topic-based (also referred to as subject-based): In this approach, we make the

assumption that each notification is expressed in terms of a number of fields, with one

field denoting the topic. Subscriptions are then defined in terms of the topic of

interest. This approach is equivalent to channel-based approaches, with the

difference that topics are implicitly defined in the case of channels but explicitly

declared as one of the fields in topic-based approaches. The expressiveness of topicbased approaches can also be enhanced by introducing hierarchical organization of

topics. For example, let us consider a publish-subscribe system for this book.

Subscriptions could be defined in terms of indirect\_communication or

indirect\_communication/publish-subscribe. Subscribers expressing interest in the

former will receive all events related to this chapter, whereas with the latter

subscribers can instead express an interest in the more specific topic of publishsubscribe.

Figure 6.8 The publish-subscribe paradigm

Publishers Subscribers

publish(e1)

subscribe(t2)

subscribe(t1)

publish(e2)

advertise(t1)

notify(e1)

Publish-subscribe systemSECTION 6.3 PUBLISH-SUBSCRIBE SYSTEMS 263

Content-based: Content-based approaches are a generalization of topic-based

approaches allowing the expression of subscriptions over a range of fields in an event

notification. More specifically, a content-based filter is a query defined in terms of

compositions of constraints over the values of event attributes. For example, a

subscriber could express interest in events that relate to the topic of publish-subscribe

systems, where the system in question is the ‘CORBA Event Service’ and where the

author is ‘Tim Kindberg’ or ‘Gordon Blair’. The sophistication of the associated

query languages varies from system to system, but in general this approach is

significantly more expressive than channel- or topic-based approaches, but with

significant new implementation challenges (discussed below).

Type-based: These approaches are intrinsically linked with object-based approaches

where objects have a specified type. In type-based approaches, subscriptions are

defined in terms of types of events and matching is defined in terms of types or

subtypes of the given filter. This approach can express a range of filters, from coarsegrained filtering based on overall type names to more fine-grained queries defining

attributes and methods of a given object. Such fine-grained filters are similar in

expressiveness to content-based approaches. The advantages of type-based

approaches are that they can be integrated elegantly into programming languages and

they can check the type correctness of subscriptions, eliminating some kinds of

subscription errors.

As well as these classical categories, a number of commercial systems are based on

subscribing directly to objects of interest. Such systems are similar to type-based

approaches in that they are intrinsically linked to object-based approaches, although

they differ by focusing on changes of state of the objects of interest rather than

predicates associated with the types of objects. They allow one object to react to a

change occurring in another object. Notifications of events are asynchronous and

determined by their receivers. In particular, in interactive applications, the actions that

the user performs on objects – for example, by manipulating a button with the mouse or

entering text in a text box via the keyboard – are seen as events that cause changes in the

objects that maintain the state of the application. The objects that are responsible for

displaying a view of the current state are notified whenever the state changes.

Rosenblum and Wolf [1997] describe a general architecture for this style of

publish-subscribe system. The main component in their architecture is an event service

that maintains a database of event notifications and of interests of subscribers. The event

service is notified of events that occur at objects of intetest. Subscribers inform the event

service about the types of events they are interested in. When an event occurs at an

object of interest a message containing the notification is sent directly to the subscribers

of that type of event.

The Jini distributed event specification described by Arnold et al. [1999] is one

leading example of this approach, and a case study on Jini, together with further

background information on this style of approach, can be found on the companion web

site for the book [www.cdk5.net/rmi]. Note, however, that Jini is a relatively primitive

example of a distributed event-based system that allows direct connectivity between

producers and consumers of events (hence compromising time and space uncoupling).

A number of more experimental approaches are also being investigated. For

example, some researchers are considering the added expressiveness of context [Frey264 CHAPTER 6 INDIRECT COMMUNICATION

and Roman 2007, Meier and Cahill 2010]. Context and context-awareness are major

concepts in mobile and ubiquitous computing. Context is defined in Chapter 19 as an

aspect of the physical circumstances of relevance to the system behaviour. One intuitive

example of context is location, and such systems have the potential for users to subscribe

to events associated with a given location – for example, any emergency messages

associated with the building where a user is located. Cilia et al. [2004] have also

introduced concept-based subscription models whereby filters are expressed in terms of

the semantics as well as the syntax of events. More specifically, data items have an

associated semantic context that captures the meaning of those items, allowing for

interpretation and possible translation into different data formats, thus addressing

heterogeneity.

For some classes of application, such as the financial trading system described in

Chapter 1, it is not enough for subscriptions to express queries over individual events.

Rather, there is a need for more complex systems that can recognize complex event

patterns. For example, Chapter 1 introduced the example of buying and selling shares

based on observing temporal sequences of events related to share prices, demonstrating

the need for complex event processing (or composite event detection, as it is sometimes

called). Complex event processing allows the specification of patterns of events as they

occur in the distributed environment – for example, ‘inform me if water levels rise by at

least 20% in the River Eden in at least three places and simulation models are also

reporting a risk of flooding’. A further example of an event pattern arose in Chapter 1,

concerned with detecting share price movements over a given time period. In general,

patterns can be logical, temporal or spatial. For further information on complex event

processing, refer to Muhl et al. [2006].

6.3.2 Implementation issues

From the description above, the task of a publish-subscribe system is clear: to ensure that

events are delivered efficiently to all subscribers that have filters defined that match the

event. Added to this, there may be additional requirements in terms of security,

scalability, failure handling, concurrency and quality of service. This makes the

implementation of publish-subscribe systems rather complex, and this has been an area

of intense investigation in the research community. We consider key implementation

issues below, examining centralized versus distributed implementations before moving

on to consider the overall system architecture required to implement publish-subscribe

systems (particularly distributed implementations of content-based approaches). We

conclude the section by summarizing the design space of publish-subscribe systems,

with associated pointers to the literature.

Centralized versus distributed implementations • A number of architectures for the

implementation of publish-subscribe systems have been identified. The simplest

approach is to centralize the implementation in a single node with a server on that node

acting as an event broker. Publishers then publish events (and optionally send

advertisements) to this broker, and subscribers send subscriptions to the broker and

receive notifications in return. Interaction with the broker is then through a series of

point-to-point messages; this can be implemented using message passing or remote

invocation.SECTION 6.3 PUBLISH-SUBSCRIBE SYSTEMS 265

This approach is straightforward to implement, but the design lacks resilience and

scalability, since the centralized broker represents a single point for potential system

failure and a performance bottleneck. Consequently, distributed implementations of

publish-subscribe systems are also available. In such schemes, the centralized broker is

replaced by a network of brokers that cooperate to offer the desired functionality as

illustrated in Figure 6.9. Such approaches have the potential to survive node failure and

have been shown to be able to operate well in Internet-scale deployments.

Taking this a step further, it is possible to have a fully peer-to-peer

implementation of a publish-subscribe system. This is a very popular implementation

strategy for recent systems. In this approach, there is no distinction between publishers,

subscribers and brokers; all nodes act as brokers, cooperatively implementing the

required event routing functionality (as discussed further below).

Overall systems architecture • As mentioned above, the implementation of centralized

schemes is relatively straightforward, with the central service maintaining a repository

of subscriptions and matching event notifications with this set of subscriptions.

Similarly, the implementations of channel-based or topic-based schemes are relatively

straightforward. For example, a distributed implementation can be achieved by mapping

channels or topics onto associated groups (as defined in Section 6.2) and then using the

underlying multicast communication facilities to deliver events to interested parties

(using reliable and ordered variants, as appropriate). The distributed implementation of

content-based (or by extrapolation, type-based) approaches is more complex and

deserving of further consideration. The range of architectural choices for such

approaches is captured in Figure 6.10 (adapted from Baldoni and Virgillito [2005]).

In the bottom layer, publish-subscribe systems make use of a range of interprocess

communication services, such as TCP/IP, IP multicast (where available) or more

specialized services, as offered for example by wireless networks. The heart of the

Figure 6.9 A network of brokers

Broker network

P1

P2

P3

Publishers Subscribers

S1

S2

S3266 CHAPTER 6 INDIRECT COMMUNICATION

architecture is provided by the event routing layer supported by a network overlay

infrastructure. Event routing performs the task of ensuring that event notifications are

routed as efficiently as possible to appropriate subscribers, whereas the overlay

infrastructure supports this by setting up appropriate networks of brokers or peer-to-peer

structures. For content-based approaches, this problem is referred to as content-based

routing (CBR), with the goal being to exploit content information to efficiently route

events to their required destination. The top layer implements matching – that is,

ensuring that events match a given subscription. While this can be implemented as a

discrete layer, often matching is pushed down into the event routing mechanisms, as will

become apparent shortly.

Within this overall architecture, there is a wide variety of implementation

approaches. We step through a select set of implementations to illustrate the general

principles behind content-based routing:

Flooding: The simplest approach is based on flooding, that is, sending an event

notification to all nodes in the network and then carrying out the appropriate

matching at the subsciber end. As an alternative, flooding can be used to send

subscriptions back to all possible publishers, with the matching carried out at the

publishing end and matched events sent directly to the relevant subscribers using

point-to-point communication. Flooding can be implemented using an underlying

broadcast or multicast facility. Alternatively, brokers can be arranged in an acyclic

graph in which each forwards incoming event notifications to all its neighbours

(effectively providing a multicast overlay, as discussed in Section 4.5.1). This

approach has the benefit of simplicity but can result in a lot of unnecessary network

Figure 6.10 The architecture of publish-subscribe systems

Network protocols

Matching

Event routing

Overlay networks

TCP/IP IP mcast 802.11g MAC bcast

Flooding Filtering Rendezvous Informed

gossip

Broker

network

multicast Group DHT Gossip

Publish-subscribe architectureSECTION 6.3 PUBLISH-SUBSCRIBE SYSTEMS 267

traffic. Hence, the alternative schemes described below try to optimize the number of

messages exchanged through consideration of content.

Filtering: One principle that underpins many approaches is to apply filtering in the

network of brokers. This is referred to as filtering-based routing. Brokers forward

notifications through the network only where there is a path to a valid subscriber.

This is achieved by propagating subscription information through the network

towards potential publishers and then storing associated state at each broker. More

specifically, each node must maintain a neighbours list containing a list of all

connected neighbours in the network of brokers, a subscription list containing a list

of all directly connected subscribers serviced by this node, and a routing table.

Crucially, this routing table maintains a list of neighbours and valid subscriptions for

that pathway.

This approach also requires an implementation of matching on each node in the

network of brokers: in particular, a match function takes a given event notification

and a list of nodes together with associated subscriptions and returns a set of nodes

where the notification matches the subscription. The specific algorithm for this

filtering approach is captured in Figure 6.11 (taken from Baldoni and Virgillito

[2005]). When a broker receives a publish request from a given node, it must pass

this notification to all connected nodes where there is a corresponding matching

subscription and also decide where to propagate this event through the network of

brokers. Lines 2 and 3 achieve the first goal by matching the event against the

subscription list and then forwarding the event to all the nodes with matching

subscriptions (the matchlist). Lines 4 and 5 then use the match function again, this

time matching the event against the routing table and forwarding only to the paths

that lead to a subscription (the fwdlist). Brokers must also deal with incoming

subscription events. If the subscription event is from an immediately connected

subscriber, then this subscription must be entered in the subscriptions table (lines 7

and 8). Otherwise, the broker is an intermediary node; this node now knows that a

pathway exists towards this subscription and hence an appropriate entry is added to

the routing table (line 9). In both cases, this subscription event is then passed to all

neighbours apart from the originating node (line 10).

Figure 6.11 Filtering-based routing

upon receive publish(event e) from node x 1

matchlist := match(e, subscriptions) 2

send notify(e) to matchlist; 3

fwdlist := match(e, routing); 4

send publish(e) to fwdlist - x; 5

upon receive subscribe(subscription s) from node x 6

if x is client then 7

add x to subscriptions; 8

else add(x, s) to routing; 9

send subscribe(s) to neighbours - x; 10268 CHAPTER 6 INDIRECT COMMUNICATION

Advertisements: The pure filtering-based approach described above can generate a

lot of traffic due to propagation of subscriptions, with subscriptions essentially using

a flooding approach back towards all possible publishers. In systems with

advertisements this burden can be reduced by propagating the advertisements

towards subscribers in a similar (actually, symmetrical) way to the propagation of

subscriptions. There are interesting trade-offs between the two approaches, and some

systems adopt both approaches in tandem [Carzaniga et al. 2001].

Rendezvous: Another approach to control the propagation of subscriptions (and to

achieve a natural load balancing) is the rendezvous approach. To understand this

approach, it is necessary to view the set of all possible events as an event space and

to partition responsibility for this event space between the set of brokers in the

network. In particular, this approach defines rendezvous nodes, which are broker

nodes responsible for a given subset of the event space. To achieve this, a given

rendezvous-based routing algorithm must define two functions. First, SN(s) takes a

given subscription, s, and returns one or more rendezvous nodes that take

responsibility for that subscription. Each such rendezvous node maintains a

subscription list as in the filtering approach above, and forwards all matching events

to the set of subscribing nodes. Second, when an event e is published, the function

EN(e) also returns one or more rendezvous nodes, this time responsible for matching

e against subscriptions in the system. Note that both SN(s) and EN(e) return more

than one node if reliability is a concern. Note also that this approach only works if

the intersection of EN(e) and SN(s) is non-empty for a given e that matches s (known

as the mapping intersection rule, as defined by Baldoni and Virgillito [2005]). The

corresponding code for rendezvous-based routing is shown in Figure 6.12 (again

taken from Baldoni and Virgillito [2005]). This time, we leave the interpretation of

the algorithm as an exercise for the reader (see Exercise 6.11).

One interesting interpretation of rendezvous-based routing is to map the event

space onto a distributed hash table (DHT). Distributed hash tables were introduced

briefly in Section 4.5.1 and are examined in more detail in Chapter 10. A distributed

Figure 6.12 Rendezvous-based routing

upon receive publish(event e) from node x at node i

rvlist := EN(e);

if i in rvlist then begin

matchlist <- match(e, subscriptions);

send notify(e) to matchlist;

end

send publish(e) to rvlist - i;

upon receive subscribe(subscription s) from node x at node i

rvlist := SN(s);

if i in rvlist then

add s to subscriptions;

else

send subscribe(s) to rvlist - i;SECTION 6.3 PUBLISH-SUBSCRIBE SYSTEMS 269

hash table is a style of network overlay that distributes a hash table over a set of nodes

in a peer-to-peer network. The key observation for rendezvous-based routing is that the

hash function can be used to map both events and subscriptions onto a corresponding

rendezvous node for the management of such subscriptions.

It is possible to employ other peer-to-peer middleware approaches to underpin

event routing in publish-subscribe systems. Indeed, this is a very active area of research

with many novel and interesting proposals emerging, particularly for very large-scale

systems [Carzaniga et al. 2001]. One specific approach is to adopt gossiping as a means

of supporting event routing. Gossip-based approaches are a popular mechanism for

achieving multicast (including reliable multicast), as discussed in Section 18.4.1. They

operate by nodes in the network periodically and probabilistically exchanging events (or

data) with neighbouring nodes. Through this approach, it is possible to propagate events

effectively through the network without the structure imposed by other approaches. A

pure gossip approach is effectively an alternative strategy for implementing flooding, as

described above. However, it is possible to take into account local information and, in

particular, content to achieve what is referred to as informed gossip. Such approaches

can be particularly attractive in highly dynamic environments where network or node

churn can be high [Baldoni et al. 2005].

6.3.3 Examples of publish-subscribe systems

We conclude this section by listing some major examples of publish-subscribe systems,

providing references for further reading (see Figure 6.13). This figure also captures the

design space for publish-subscribe systems, illustrating how different designs can result

from decisions on subscription and distribution models and, especially, the underlying

event routing strategy. Note that event routing is not required for centralized schemes,

hence the blank entry in the table.

Figure 6.13 Example publish-subscribe systems

System (and further reading) Subscription

model

Distribution

model

Event routing

CORBA Event Service (Chapter 8) Channel-based Centralized -

TIB Rendezvouz [Oki et al. 1993] Topic-based Distributed Ffiltering

Scribe [Castro et al. 2002b] Topic-based Peer-to-peer

(DHT)

Rendezvous

TERA [Baldoni et al. 2007] Topic-based Peer-to-peer Informed gossip

Siena [Carzaniga et al. 2001] Content-based Distributed Filtering

Gryphon [www.research.ibm.com] Content-based Distributed Filtering

Hermes [Pietzuch and Bacon 2002] Topic- and

content-based

Distributed Rendezvous and

filtering

MEDYM [Cao and Singh 2005] Content-based Distributed Flooding

Meghdoot [Gupta et al. 2004] Content-based Peer-to-peer Rendezvous

Structure-less CBR [Baldoni et al. 2005] Content-based Peer-to-peer Informed gossip270 CHAPTER 6 INDIRECT COMMUNICATION

6.4 Message queues

Message queues (or more accurately, distributed message queues) are a further

important category of indirect communication systems. Whereas groups and publishsubscribe provide a one-to-many style of communication, message queues provide a

point-to-point service using the concept of a message queue as an indirection, thus

achieving the desired properties of space and time uncoupling. They are point-to-point

in that the sender places the message into a queue, and it is then removed by a single

process. Message queues are also referred to as Message-Oriented Middleware. This is

a major class of commercial middleware with key implementations including IBM’s

WebSphere MQ, Microsoft’s MSMQ and Oracle’s Streams Advanced Queuing (AQ).

The main use of such products is to achieve Enterprise Application Integration (EAI) –

that is, integration between applications within a given enterprise – a goal that is

achieved by the inherent loose coupling of message queues. They are also extensively

used as the basis for commercial transaction processing systems because of their

intrinsic support for transactions, discussed further in Section 6.4.1.

We examine message queues in more detail below, considering the programming

model offered by message queueing systems before addressing implementation issues.

The section then concludes by presenting the Java Messaging Service (JMS) as an

example of a middleware specification supporting message queues (and also publishsubscribe).

6.4.1 The programming model

The programming model offered by message queues is very simple. It offers an

approach to communication in distributed systems through queues. In particular,

producer processes can send messages to a specific queue and other (consumer)

processes can then receive messages from this queue. Three styles of receive are

generally supported:

• a blocking receive, which will block until an appropriate message is available;

• a non-blocking receive (a polling operation), which will check the status of the

queue and return a message if available, or a not available indication otherwise;

• a notify operation, which will issue an event notification when a message is

available in the associated queue.

This overall approach is captured pictorially in Figure 6.14.

A number of processes can send messages to the same queue, and likewise a

number of receivers can remove messages from a queue. The queuing policy is normally

first-in-first-out (FIFO), but most message queue implementations also support the

concept of priority, with higher-priority messages delivered first. Consumer processes

can also select messages from the queue based on properties of a message. In more

detail, a message consists of a destination (that is, a unique identifier designating the

destination queue), metadata associated with the message, including fields such as the

priority of the message and the delivery mode, and also the body of the message. The

body is normally opaque and untouched by the message queue system. The associatedSECTION 6.4 MESSAGE QUEUES 271

content is serialized using any of the standard approaches described in Section 4.3; that

is, marshalled data types, object serialization or XML structured messages. Message

sizes are configurable and can be very large – for example, on the order of a 100 Mbytes

Given the fact that message bodies are opaque, message selection is normally expressed

through predicates defined over the metadata.

Oracle’s AQ introduces an interesting twist on this basic idea to achieve better

integration with (relational) databases; in Oracle AQ, messages are rows in a database

table, and queues are database tables that can be queried using the full power of a

database query language.

One crucial property of message queue systems is that messages are persistent –

that is, message queues will store the messages indefinitely (until they are consumed)

and will also commit the messages to disk to enable reliable delivery. In particular,

following the definition of reliable communication in Section 2.4.2, any message sent is

eventually received (validity) and the message received is identical to the one sent, and

no messages are delivered twice (integrity). Message queue systems therefore guarantee

that messages will be delivered (and delivered once) but cannot say anything about the

timing of the delivery.

Message passing systems can also support additional functionality:

• Most commercially available systems provide support for the sending or receiving

of a message to be contained within a transaction. The goal is to ensure that all the

steps in the transaction are completed, or the transaction has no effect at all (the

‘all or nothing’ property). This relies on interfacing with an external transaction

service, provided by the middleware environment. Detailed consideration of

transactions is deferred until Chapter 16.

• A number of systems also support message transformation, whereby an arbitrary

transformation can be performed on an arriving message. The most common

application of this concept is to transform messages between formats to deal with

heterogeneity in underlying data representations. This could be as simple as

Figure 6.14 The message queue paradigm

Producers Message queue system

Poll

Message

Send

Receive

Consumers

Notify

..

Send

Send272 CHAPTER 6 INDIRECT COMMUNICATION

transforming from one byte order to another (big-endian to little-endian) or more

complex, involving for example a transformation from one external data

representation to another (such as SOAP to IIOP). Some systems also allow

programmers to develop their own application-specific transformation in response

to triggers from the underlying message queuing system. Message transformation

is an important tool in dealing with heterogeneity generally and achieving

Enterprise Application Integration in particular (as discussed above). Note that the

term message broker is often used to denote a service responsible for message

transformation.

• Some message queue implementations also provide support for security. For

example, WebSphere MQ provides support for the confidential transmission of

data using the Secure Sockets Layer (SSL) together with support for

authentication and access control. See Chapter 11.

As a final word on the programming abstraction offered by message queues, it is helpful

to compare the style of programming with other communication paradigms. Message

queues are similar in many ways to the message-passing systems considered in Chapter

4. The difference is that whereas message-passing systems have implicit queues

associated with senders and receivers (for example, the message buffers in MPI),

message queuing systems have explicit queues that are third-party entities, separate

from the sender and the receiver. It is this key difference that makes message queues an

indirect communication paradigm with the crucial properties of space and time

uncoupling.

6.4.2 Implementation issues

The key implementation issue for message queuing systems is the choice between

centralized and distributed implementations of the concept. Some implementations are

centralized, with one or more message queues managed by a queue manager located at

a given node. The advantage of this scheme is simplicity, but such managers can become

rather heavyweight components and have the potential to become a bottleneck or a

single point of failure. As a result, more distributed implementations have been

proposed. To illustrate distributed architectures, we briefly consider the approach

adopted in WebSphere MQ as representative of the state-of-the-art in this area.

Case study: WebSphere MQ • WebSphere MQ is middleware developed by IBM based

on the concept of message queues, offering an indirection between senders and receivers

of messages [www.redbooks.ibm.com]. Queues in WebSphere MQ are managed by

queue managers which host and manage queues and allow applications to access queues

through the Message Queue Interface (MQI). The MQI is a relatively simple interface

allowing applications to carry out operations such as connecting to or disconnecting

from a queue (MQCONN and MQDISC) or sending/receiving messages to/from a queue

(MQPUT and MQGET). Multiple queue managers can reside on a single physical server.

Client applications accessing a queue manager may reside on the same physical

server. More generally, though, they will be on different machines and must then

communicate with the queue manager through what is known as a client channel. Client

channels adopt the rather familiar concept of a proxy, as introduced in Chapters 2 and 5,SECTION 6.4 MESSAGE QUEUES 273

whereby MQI commands are issued on the proxy and then sent transparently to the

queue manager for execution using RPC. An example of such a configuration is shown

in Figure 6.15. In this configuration, a client application is sending messages to a remote

queue manager and multiple services (on the same machine as the server) are then

consuming the incoming messages.

This is a very simple use of WebSphere MQ, and in practice it is more common

for queue managers to be linked together into a federated structure, mirroring the

approach often adopted in publish-subscribe systems (with networks of brokers). To

achieve this, MQ introduces the concept of a message channel as a unidirectional

connection between two queue managers that is used to forward messages

asynchronously from one queue to another. Note the terminology here: a message

channel is a connection between two queue managers, whereas a client channel is a

connection between a client application and a queue manager. A message channel is

managed by a message channel agent (MCA) at each end. The two agents are

responsible for establishing and maintaining the channel, including an initial negotiation

to agree on the properties of the channel (including security properties). Routing tables

are also included in each queue manager, and together with channels this allows

arbitrary topologies to be created.

This ability to create customized topologies is crucial to WebSphere MQ,

allowing users to determine the right topology for their application domain, for example

to deliver certain requirements in terms of scalability and performance. Tools are

provided for systems administrators to create suitable topologies and to hide the

complexities of establishing message channels and routing strategies.

A wide range of topologies can be created, including trees, meshes or a bus-based

configuration. To illustrate the concept of topologies further, we present one example

topology often used in WebSphere MQ deployments, the hub-and-spoke topology.

Figure 6.15 A simple networked topology in WebSphere MQ

Queue manager

T

Proxy

Stub

Client channel

Client

Services274 CHAPTER 6 INDIRECT COMMUNICATION

The hub-and-spoke approach: In the hub-and-spoke topology, one queue manager is

designated as the hub. The hub hosts a range of services. Client applications do not

connect directly to this hub but rather connect through queue managers designated as

spokes. Spokes relay messages to the message queue of the hub for processing by the

various services. Spokes are placed strategically around the network to support different

clients. The hub is placed somewhere appropriate in the network, on a node with

sufficient resources to deal with the volume of traffic. Most applications and services are

located on the hub, although it is also possible to have some more local services on

spokes.

This topology is heavily used with WebSphere MQ, particularly in large-scale

deployments covering significant geographical areas (and possibly crossing

organizational boundaries). The key to the approach is to be able to connect to a local

spoke over a high-bandwidth connection, for example over a local area network (spokes

may even be placed in the same physical machine as client applications to minimize

latency).

Recall that communication between a client application and a queue manager uses

RPC, whereas internal communication between queue managers is asynchronous (nonblocking). This means that the client application is only blocked until the message is

deposited in the local queue manager (the local spoke); subsequent delivery, potentially

over wide area networks, is asynchronous but guaranteed to be reliable by the

WebSphere MQ middleware.

Clearly, the drawback of this architecture is that the hub can be a potential

bottleneck and a single point of failure. WebSphere MQ also supports other facilities to

overcome these problems, including queue manager clusters, which allow multiple

instances of the same service to be supported by multiple queue managers with implicit

load balancing across the different instantiations [www.redbooks.ibm.com].

6.4.3 Case study: The Java Messaging Service (JMS)

The Java Messaging Service (JMS) [java.sun.com XI] is a specification of a

standardized way for distributed Java programs to communicate indirectly. Most

notably, as will be explained, the specification unifies the publish-subscribe and

message queue paradigms at least superficially by supporting topics and queues as

alternative destinations of messages. A wide variety of implementations of the common

specification are now available, including Joram from OW2, Java Messaging from

JBoss, Sun’s Open MQ, Apache ActiveMQ and OpenJMS. Other platforms, including

WebSphere MQ, also provide a JMS interface on to their underlying infrastructure.

JMS distinguishes between the following key roles:

• A JMS client is a Java program or component that produces or consumes

messages, a JMS producer is a program that creates and produces messages and a

JMS consumer is a program that receives and consumes messages.

• A JMS provider is any of the multiple systems that implement the JMS

specification.

• A JMS message is an object that is used to communicate information between

JMS clients (from producers to consumers).SECTION 6.4 MESSAGE QUEUES 275

• A JMS destination is an object supporting indirect communication in JMS. It is

either a JMS topic or a JMS queue.

Programming with JMS • The programming model offered by the JMS API is captured

in Figure 6.16. To interact with a JMS provider, it is first necessary to create a

connection between a client program and the provider. This is created through a

connection factory (a service responsible for creating connections with the required

properties). The resultant connection is a logical channel between the client and

provider; the underlying implementation may, for example, map onto a TCP/IP socket

if implemented over the Internet. Note that two types of connection can be established,

a TopicConnection or a QueueConnection, thus enforcing a clear separation between the

two modes of operation within given connections.

Connections can be used to create one or more sessions – a session is a series of

operations involving the creation, production and consumption of messages related to a

logical task. The resultant session object also supports operations to create transactions,

supporting all-or-nothing execution of a series of operations, as discussed in Section

6.4.1. There is a clear distinction between topic sessions and queue sessions in that a

TopicConnection can support one or more topic sessions and a QueueConnection can

support one or more queue sessions, but it is not possible to mix session styles in a

connection. Thus, the two styles of operation are integrated in a rather superficial way.

The session object is central to the operation of JMS, supporting methods for the

creation of messages, message producers and message consumers:

• In JMS, a message consists of three parts: a header, a set of properties and the

body of the message. The header contains all the information needed to identify

and route the message, including the destination (a reference to either a topic or a

Figure 6.16 The programming model offered by JMS

Connection factory

Connection

Message Session Message

producer consumer

Creates

Destination :

Topic

Queue

Destination :

Topic

Queue

Message

Sends to Receives from

Communicates276 CHAPTER 6 INDIRECT COMMUNICATION

queue), the priority of the message, the expiration date, a message ID and a

timestamp. Most of these fields are created by the underlying system, but some

can be filled in specifically through the associated constructor methods. Properties

are all user-defined and can be used to associate other application-specific

metadata elements with a message. For example, if implementing a context-aware

system (as discussed in Chapter 19), the properties can be used to express

additional context associated with the message, including a location field. As in

the general description of message queue systems, this body is opaque and

untouched by the system. In JMS, the body can be any one of a text message, a

byte stream, a serialized Java object, a stream of primitive Java values or a more

structured set of name/value pairs.

• A message producer is an object used to publish messages under a particular topic

or to send messages to a queue.

• A message consumer is an object used to subscribe to messages concerned with a

given topic or to receive messages from a queue. The consumer is more

complicated than the producer, for two reasons. First, it is possible to associate

filters with message consumers by specifying what is known as a message selector

– a predicate defined over the values in the header and properties parts of a

message (not the body). A subset of the database query language SQL is used to

specify properties. This could be used, for example, to filter messages from a

Figure 6.17 Java class FireAlarmJMS

import javax.jms.\*;

import javax.naming.\*;

public class FireAlarmJMS {

public void raise() {

try { 1

Context ctx = new InitialContext(); 2

TopicConnectionFactory topicFactory = 3

(TopicConnectionFactory)ctx.lookup("TopicConnectionFactory"); 4

Topic topic = (Topic)ctx.lookup("Alarms"); 5

TopicConnection topicConn = 6

topicConnectionFactory.createTopicConnection(); 7

TopicSession topicSess = topicConn.createTopicSession(false, 8

Session.AUTO\_ACKNOWLEDGE); 9

TopicPublisher topicPub = topicSess.createPublisher(topic); 10

TextMessage msg = topicSess.createTextMessage(); 11

msg.setText("Fire!"); 12

topicPub.publish(message); 13

} catch (Exception e) { 14

} 15

}SECTION 6.4 MESSAGE QUEUES 277

given location in the context-aware example above. Second, there are two modes

provided for receiving messages: the program either can block using a receive

operation or it can establish a message listener object which must provide an

onMessage method that is invoked whenever a suitable message is identified.

A simple example • To illustrate the use of JMS, we return to our example of Section

6.2.3, the fire alarm service, and show how this would be implemented in JMS. We

choose the topic-based publish-subscribe service as this is intrinsically a one-to-many

application, with the alarm producing alarm messages targeted towards many consumer

applications.

The code for the fire alarm object is shown in Figure 6.17. It is more complicated

than the equivalent JGroups example mainly because of the need to create a connection,

session, publisher and message, as shown in lines 6–11. This is relatively

straightforward apart from the parameters of createTopicSession, which are whether the

session should be transactional (false in this case) and the mode of acknowledging

messages (AUTO\_ACKNOWLEDGE in this example, which means a session

automatically acknowledges the receipt of a message). There is additional complexity

associated with finding the connection factory and topic in the distributed environment

(the complexity of connecting to a named channel in JGroups is all hidden in the connect

method). This is achieved using JNDI (the Java Naming and Directory Interface) in lines

2 to 5. This is included for completeness and it is assumed that readers can appreciate

Figure 6.18 Java class FireAlarmConsumerJMS

import javax.jms.\*;

import javax.naming.\*;

public class FireAlarmConsumerJMS {

public String await() {

try { 1

Context ctx = new InitialContext(); 2

TopicConnectionFactory topicFactory = 3

(TopicConnectionFactory)ctx.lookup("TopicConnectionFactory"); 4

Topic topic = (Topic)ctx.lookup("Alarms"); 5

TopicConnection topicConn = 6

topicConnectionFactory.createTopicConnection(); 7

TopicSession topicSess = topicConn.createTopicSession(false, 8

Session.AUTO\_ACKNOWLEDGE); 9

TopicSubscriber topicSub = topicSess.createSubscriber(topic); 10

topicSub.start(); 11

TextMessage msg = (TextMessage) topicSub.receive(); 12

return msg.getText(); 13

} catch (Exception e) { 14

return null; 15

} 16

}278 CHAPTER 6 INDIRECT COMMUNICATION

the purpose of these lines of code without further explanation. Lines 12 and 13 contain

the crucial code to create a new message and then publish it to the appropriate topic. The

code to create a new instance of the FireAlarmJMS class and then raise an alarm is:

FireAlarmJMS alarm = new FireAlarmJMS();

alarm.raise();

The corresponding code for the receiver end is very similar and is shown in Figure 6.18.

Lines 2–9 are identical and create the required connection and session, respectively.

This time, though, an object of type TopicSubscriber is created next (line 10), and the

start method in line 11 starts this subscription, enabling messages to be received. The

blocking receive in line 12 then awaits an incoming message and line 13 returns the

textual contents of this message as a string. This class is used as follows by a consumer:

FireAlarmConsumerJMS alarmCall = new FireAlarmConsumerJMS();

String msg = alarmCall.await();

System.out.println("Alarm received: "+msg);

Overall this case study has illustrated how both publish-subscribe and message queues

can be supported by a single middleware solution (in this case JMS), offering the

programmer the choice of one-to-many or point-to-point variants of indirect

communication, respectively.

6.5 Shared memory approaches

In this section, we examine indirect communication paradigms that offer an abstraction

of shared memory. We look briefly at distributed shared memory techniques that were

developed principally for parallel computing before moving on to tuple space

communication, an approach that allows programmers to read and write tuples from a

shared tuple space. Whereas distributed shared memory operates at the level of reading

and writing bytes, tuple spaces offer a higher-level perspective in the form of semistructured data. In addition, whereas distributed shared memory is accessed by address,

tuple spaces are associative, offering a form of content-addressable memory [Gelernter

1985].

Chapter 18 of the fourth edition of this book provided in-depth coverage of

distributed shared memory, including consistency models and several case studies. This

chapter can be found on the companion web site for the book [www.cdk5.net/dsm].

6.5.1 Distributed shared memory

Distributed shared memory (DSM) is an abstraction used for sharing data between

computers that do not share physical memory. Processes access DSM by reads and

updates to what appears to be ordinary memory within their address space. However, an

underlying runtime system ensures transparently that processes executing at different

computers observe the updates made by one another. It is as though the processes access

a single shared memory, but in fact the physical memory is distributed (see Figure 6.19).SECTION 6.5 SHARED MEMORY APPROACHES 279

The main point of DSM is that it spares the programmer the concerns of message

passing when writing applications that might otherwise have to use it. DSM is primarily

a tool for parallel applications or for any distributed application or group of applications

in which individual shared data items can be accessed directly. DSM is in general less

appropriate in client-server systems, where clients normally view server-held resources

as abstract data and access them by request (for reasons of modularity and protection).

Message passing cannot be avoided altogether in a distributed system: in the

absence of physically shared memory, the DSM runtime support has to send updates in

messages between computers. DSM systems manage replicated data: each computer has

a local copy of recently accessed data items stored in DSM, for speed of access. The

problems of implementing DSM are related to the replication issues to be discussed in

Chapter 18, as well as to those of caching shared files, discussed in Chapter 12.

One of the first notable examples of a DSM implementation was the Apollo

Domain file system [Leach et al. 1983], in which processes hosted by different

workstations share files by mapping them simultaneously into their address spaces. This

example shows that distributed shared memory can be persistent. That is, it may outlast

the execution of any process or group of processes that accesses it and be shared by

different groups of processes over time.

The significance of DSM first grew alongside the development of shared-memory

multiprocessors (discussed further in Section 7.3). Much research has gone into

investigating algorithms suitable for parallel computation on these multiprocessors. At

the hardware architectural level, developments include both caching strategies and fast

processor-memory interconnections, aimed at maximizing the number of processors

that can be sustained while achieving fast memory access latency and throughput

[Dubois et al. 1988]. Where processes are connected to memory modules over a

common bus, the practical limit is on the order of 10 processors before performance

degrades drastically due to bus contention. Processors sharing memory are commonly

constructed in groups of four, sharing a memory module over a bus on a single circuit

board. Multiprocessors with up to 64 processors in total are constructed from such

boards in a Non-Uniform Memory Access (NUMA) architecture. This is a hierarchical

Figure 6.19 The distributed shared memory abstraction

Physical

memory

Process

accessing DSM

DSM appears as

memory in address

space of process

Physical

memory

Physical

memory

Distributed shared memory

Mappings280 CHAPTER 6 INDIRECT COMMUNICATION

architecture in which the four-processor boards are connected using a high-performance

switch or higher-level bus. In a NUMA architecture, processors see a single address

space containing all the memory of all the boards. But the access latency for on-board

memory is less than that for a memory module on a different board – hence the name of

this architecture.

In distributed-memory multiprocessors and clusters of off-the-shelf computing

components (again, see Section 7.3), the processors do not share memory but are

connected by a very high speed network. These systems, like general-purpose

distributed systems, can scale to much greater numbers of processors than a sharedmemory multiprocessor’s 64 or so. A central question that has been pursued by the DSM

and multiprocessor research communities is whether the investment in knowledge of

shared memory algorithms and the associated software can be directly transferred to a

more scalable distributed memory architecture.

Message passing versus DSM • As a communication mechanism, DSM is comparable

with message passing rather than with request-reply-based communication, since its

application to parallel processing, in particular, entails the use of asynchronous

communication. The DSM and message-passing approaches to programming can be

contrasted as follows:

Service offered: Under the message-passing model, variables have to be marshalled

from one process, transmitted and unmarshalled into other variables at the receiving

process. By contrast, with shared memory the processes involved share variables

directly, so no marshalling is necessary – even of pointers to shared variables – and

thus no separate communication operations are necessary. Most implementations

allow variables stored in DSM to be named and accessed similarly to ordinary

unshared variables. In favour of message passing, on the other hand, is that it allows

processes to communicate while being protected from one another by having private

address spaces, whereas processes sharing DSM can, for example, cause one another

to fail by erroneously altering data. Furthermore, when message passing is used

between heterogeneous computers, marshalling takes care of differences in data

representation; but how can memory be shared between computers with, for example,

different integer representations?

Synchronization between processes is achieved in the message model through

message passing primitives themselves, using techniques such as the lock server

implementation discussed in Chapter 16. In the case of DSM, synchronization is via

normal constructs for shared-memory programming such as locks and semaphores

(although these require different implementations in the distributed memory

environment). Chapter 7 briefly discusses such synchronization objects in the context

of programming with threads.

Finally, since DSM can be made persistent, processes communicating via DSM

may execute with non-overlapping lifetimes. A process can leave data in an agreed

memory location for the other to examine when it runs. By contrast, processes

communicating via message passing must execute at the same time.

Efficiency: Experiments show that certain parallel programs developed for DSM can

be made to perform about as well as functionally equivalent programs written for

message-passing platforms on the same hardware [Carter et al. 1991] – at least in theSECTION 6.5 SHARED MEMORY APPROACHES 281

case of relatively small numbers of computers (10 or so). However, this result cannot

be generalized. The performance of a program based on DSM depends upon many

factors, as we shall discuss below – particularly the pattern of data sharing (such as

whether an item is updated by several processes).

There is a difference in the visibility of costs associated with the two types of

programming. In message passing, all remote data accesses are explicit and therefore

the programmer is always aware of whether a particular operation is in-process or

involves the expense of communication. Using DSM, however, any particular read

or update may or may not involve communication by the underlying runtime support.

Whether it does or not depends upon such factors as whether the data have been

accessed before and the sharing pattern between processes at different computers.

There is no definitive answer as to whether DSM is preferable to message passing for

any particular application. DSM remains a tool whose ultimate status depends upon the

efficiency with which it can be implemented.

6.5.2 Tuple space communication

Tuple spaces were first introduced by David Gelernter from Yale University as a novel

form of distributed computing based on what he refers to as generative communication

[Gelernter 1985]. In this approach, processes communicate indirectly by placing tuples

in a tuple space, from which other processes can read or remove them. Tuples do not

have an address but are accessed by pattern matching on content (content-addressable

memory, as discussed by Gelernter [1985]). The resultant Linda programming model

has been highly influential and has led to significant developments in distributed

programming including systems such as Agora [Bisiani and Forin 1988] and, more

significantly, JavaSpaces from Sun (discussed below) and IBM’s TSpaces. Tuple space

communication has also been influential in the field of ubiquitous computing, for

reasons that are explored in depth in Chapter 19.

This section provides an examination of the tuple space paradigm as it applies to

distributed computing. We start by examining the programming model offered by tuple

spaces before briefly considering the associated implementation issues. The section then

concludes by examining the JavaSpaces specification as a case study, illustrating how

tuple spaces have evolved to embrace the object-oriented world.

The programming model • In the tuple space programming model, processes

communicate through a tuple space – a shared collection of tuples. Tuples in turn consist

of a sequence of one or more typed data fields such as <"fred", 1958>, <"sid", 1964>

and <4, 9.8, "Yes">. Any combination of types of tuples may exist in the same tuple

space. Processes share data by accessing the same tuple space: they place tuples in tuple

space using the write operation and read or extract them from tuple space using the read

or take operation. The write operation adds a tuple without affecting existing tuples in

the space. The read operation returns the value of one tuple without affecting the

contents of the tuple space. The take operation also returns a tuple, but in this case it also

removes the tuple from the tuple space.

When reading or removing a tuple from tuple space, a process provides a tuple

specification and the tuple space returns any tuple that matches that specification – as

mentioned above, this is a type of associative addressing. To enable processes to282 CHAPTER 6 INDIRECT COMMUNICATION

synchronize their activities, the read and take operations both block until there is a

matching tuple in the tuple space. A tuple specification includes the number of fields and

the required values or types of the fields. For example, take(<String, integer>) could

extract either <"fred", 1958> or <"sid", 1964>; take(<String, 1958>) would extract only

<"fred", 1958> of those two.

In the tuple space paradigm, no direct access to tuples in tuple space is allowed

and processes have to replace tuples in the tuple space instead of modifying them. Thus,

tuples are immutable. Suppose, for example, that a set of processes maintains a shared

counter in tuple space. The current count (say, 64) is in the tuple <"counter", 64>. A

process must execute code of the following form in order to increment the counter in a

tuple space myTS:

<s, count> := myTS.take(<"counter", integer>);

myTS.write(<"counter", count+1>);

A further illustration of the tuple space paradigm is given in Figure 6.20. This tuple

space contains a range of tuples representing geographical information about countries

in the United Kingdom, including populations and capital cities. The take operation

take(<String, "Scotland", String>) will match <"Capital", "Scotland", "Edinburgh">,

whereas take(<String, "Scotland", Integer>) will match <"Population", "Scotland",

5168000>. The write operation write(<"Population", "Wales, 2900000>) will insert a

new tuple in the tuple space with information on the population of Wales. Finally,

read(<"Population", String, Integer) can match the equivalent tuples for the

populations of the UK, Scotland or indeed Wales, if this operation is executed after the

corresponding write operation. One will be selected nondeterministically by the tuple

<"Capital", "Scotland", "Edinburgh">

<"Capital", "Wales", "Cardiff">

<"Capital", "England", "London">

<"Capital", "N. Ireland", "Belfast">

<"Population", "Scotland", 5168000>

<"Population", "UK", 61000000>

take(<String, "Scotland", Integer>)

write(<"Population", "Wales", 2900000>)

read(<"Population", String, Integer>)

take(<String, "Scotland", String>)

Figure 6.20 The tuple space abstractionSECTION 6.5 SHARED MEMORY APPROACHES 283

space implementation and, with this being a read operation, the tuple will remain in the

tuple space.

Note that write, read and take are known as out, rd and in in Linda; we use the

more descriptive former names throughout this book. This terminology is also used in

JavaSpaces, discussed in a case study below.

Properties associated with tuple spaces: Gelernter [1985] presents some interesting

properties associated with tuple space communication, highlighting in particular both

space and time uncoupling as discussed in Section 6.1:

Space uncoupling: A tuple placed in tuple space may originate from any number of

sender processes and may be delivered to any one of a number of potential recipients.

This property is also referred to as distributed naming in Linda.

Time uncoupling: A tuple placed in tuple space will remain in that tuple space until

removed (potentially indefinitely), and hence the sender and receiver do not need to

overlap in time.

Together, these features provide an approach that is fully distributed in space and time

and also provide for a form of distributed sharing of shared variables via the tuple space.

Gelernter [1985] also explores a range of other properties associated with the

rather flexible style of naming employed in Linda (referred to as free naming). The

interested reader is directed to Gelernter’s paper for more information on this topic.

Variations on a theme: Since the introduction of Linda, refinements have been proposed

to the original model:

• The original Linda model proposed a single, global tuple space. This is not

optimal in large systems, as it leads to the danger of unintended aliasing of tuples:

as the number of tuples in a tuple space increases, there is an increasing chance of

a read or take matching a tuple by accident. This is particularly likely when

matching on types, such as with take(<String, integer>), as mentioned above.

Given this, a number of systems have proposed multiple tuple spaces, including

the ability to dynamically create tuple spaces, introducing a degree of scoping into

the system (see, for example, the JavaSpaces case study below).

• Linda was anticipated to be implemented as a centralized entity but later systems

have experimented with distributed implementations of tuple spaces (including

strategies to provide more fault tolerance). Given the importance of this topic to

this book, we focus on this in the implementation issues subsection below.

• Researchers have also experimented with modifying or extending the operations

provided in tuple spaces and adapting the underlying semantics. One rather

interesting proposal is to unify the concepts of tuples and tuple spaces by

modelling everything as (unordered) sets – that is, tuple spaces are sets of tuples

and tuples are sets of values, which may now also include tuples. This variant is

known as Bauhaus Linda [Carriero et al. 1995].

• Perhaps most interestingly, recent implementations of tuple spaces have moved

from tuples of typed data items to data objects (with attributes), turning the tuple

space into an object space. This proposal is adopted, for example, in the influential

system JavaSpaces, discussed in more detail below.284 CHAPTER 6 INDIRECT COMMUNICATION

Implementation issues • Many of the implementations of tuple spaces adopt a

centralized solution where the tuple space resource is managed by a single server. This

has advantages in terms of simplicity, but such solutions are clearly not fault tolerant and

also will not scale. Because of this, distributed solutions have been proposed.

Replication: Several systems have proposed the use of replication to overcome the

problems identified above [Bakken and Schlichting 1995, Bessani et al. 2008, Xu and

Liskov 1989].

The proposals from Bakken and Schlichting [1995] and Bessani et al. [2008]

adopt a similar approach to replication, referred to as the state machine approach and

discussed further in Chapter 18. This approach assumes that a tuple space behaves like

a state machine, maintaining state and changing this state in response to events received

from other replicas or from the environment. To ensure consistency the replicas (i) must

start in the same state (an empty tuple space), (ii) must execute events in the same order

and (iii) must react deterministically to each event. The key second property can be

guaranteed by adopting a totally ordered multicast algorithm, as discussed in Section

6.2.2.

Xu and Liskov [1989] adopt a different approach, which optimizes the replication

strategy by using the semantics of the particular tuple space operations. In this proposal,

updates are carried out in the context of the current view (the agreed set of replicas) and

tuples are also partitioned into distinct tuple sets based on their associated logical names

(designated as the first field in the tuple). The system consists of a set of workers

carrying out computations on the tuple space, and a set of tuple space replicas. A given

physical node can contain any number of workers, replicas or indeed both; a given

worker therefore may or may not have a local replica. Nodes are connected by a

communications network that may lose, duplicate or delay messages and can deliver

messages out of order. Network partitions can also occur.

A write operation is implemented by sending a multicast message over the

unreliable communications channel to all members of the view. On receipt, members

place this tuple into their replica and acknowledge receipt. The write request is repeated

until all acknowledgements are received. For the correct operation of the protocol,

replicas must detect and acknowledge duplicate requests, but not carry out the

associated write operations.

The read operation consists of sending a multicast message to all replicas. Each

replica seeks a match and returns this match to the requesting site. The first tuple

returned is delivered as the result of the read. This may come from a local node, but

given that many workers will not have a local replica, this is not guaranteed.

The take operation is more complex because of the need to agree on the tuple to

be selected and to remove this agreed tuple from all copies. The algorithm proceeds in

two phases. In phase 1, the tuple specification is sent to all replicas, and the replica

attempts to acquired the lock on the associated tuple set to serialize take requests on the

replicas (write and read operations are unaffected by the lock); if the lock cannot be

acquired, the take request is refused. Each replica that succeeds in obtaining the lock

responds with the set of matching tuples. This step is repeated until all replicas have

accepted the request and responded. The initiating process can then select one tuple from

the intersection of all the replies and return this as the result of the take request. If it isSECTION 6.5 SHARED MEMORY APPROACHES 285

not possible to obtain a majority of locks, the replicas are asked to release their locks and

phase 1 repeats.

In phase 2, this tuple must be removed from all replicas. This is achieved by

repeated multicasts to the replicas in the view until all have acknowledged deletion. As

with write requests, it is necessary for replicas to detect repeat requests in phase 2 and

to simply send another acknowledgement without carrying out another deletion

(otherwise multiple tuples could erroneously be deleted at this stage).

The steps involved for each operation are summarized in Figure 6.21. Note that a

separate algorithm is required to manage view changes if node failures occur or the

network partitions (see Xu and Liskov [1989] for details).

This algorithm is designed to minimize delay given the semantics of the three

tuple space operations:

read operations only block until the first replica responds to the request.

take operations block until the end of phase 1, when the tuple to be deleted has been

agreed.

write operations can return immediately.

Figure 6.21 Replication and the tuple space operations [Xu and Liskov 1989]

write 1. The requesting site multicasts the write request to all members of the view;

2. On receiving this request, members insert the tuple into their replica and acknowledge this action;

3. Step 1 is repeated until all acknowledgements are received.

read 1. The requesting site multicasts the read request to all members of the view;

2. On receiving this request, a member returns a matching tuple to the requestor;

3. The requestor returns the first matching tuple received as the result of the operation (ignoring others);

4. Step 1 is repeated until at least one response is received.

take Phase 1: Selecting the tuple to be removed

1. The requesting site multicasts the take request to all members of the view;

2. On receiving this request, each replica acquires a lock on the associated tuple set and, if the lock

cannot be acquired, the take request is rejected;

3. All accepting members reply with the set of all matching tuples;

4. Step 1 is repeated until all sites have accepted the request and responded with their set of tuples and

the intersection is non-null;

5. A particular tuple is selected as the result of the operation (selected randomly from the intersection

of all the replies);

6. If only a minority accept the request, this minority are asked to release their locks and phase 1 repeats.

Phase 2: Removing the selected tuple

1. The requesting site multicasts a remove request to all members of the view citing the tuple to be

removed;

2. On receiving this request, members remove the tuple from their replica, send an acknowledgement

and release the lock;

3. Step 1 is repeated until all acknowledgements are received.286 CHAPTER 6 INDIRECT COMMUNICATION

This, though, introduces unacceptable levels of concurrency. For example, a read

operation may access a tuple that should have been deleted in the second phase of a take

operation. Therefore additional levels of concurrency control are required. In particular,

Xu and Liskov [1989] introduce the following additional constraints:

• The operations of each worker must be executed at each replica in the same order

as they were issued by the worker;.

• A write operation must not be executed at any replica until all previous take

operations issued by the same worker have completed at all replicas in the

worker's view.

A further example of using replication is provided in Chapter 19, where we present the

L2imbo approach, which uses replication to provide high availability in mobile

environments [Davies et al. 1998].

Other approaches: A range of other approaches have been employed in the

implementation of the tuple space abstraction, including partitioning of the tuple space

over a number of nodes and mapping onto peer-to-peer overlays:

• The Linda Kernel developed at the University of York [Rowstron and Wood

1996] adopts an approach in which tuples are partitioned across a range of

available tuple space servers (TSSs), as illustrated in Figure 6.22. There is no

replication of tuples; that is, there is only one copy of each tuple. The motivation

is to increase performance of the tuple space, especially for highly parallel

computation.When a tuple is placed in tuple space, a hashing algorithm is used to

select one of the tuple space servers to be used. The implementation of read or

take is slightly more complex, as a tuple specification is provided that may specify

types or values of the associated fields. The hashing algorithm uses this

Figure 6.22 Partitioning in the York Linda Kernel

Local tuple

space manager

User process

Local tuple

space manager

User process

Local tuple

space manager

User process

Local tuple

space manager

User process

Local tuple

space manager

User process

Local tuple

space manager

User process

TSS TSS TSS TSS TSSSECTION 6.5 SHARED MEMORY APPROACHES 287

specification to generate a set of possible servers that may contain matching

tuples, and a linear search must then be employed until a matching tuple is

discovered. Note that because there is only a single copy of a given tuple, the

implementation of take is greatly simplified.

• Some implementations of tuple spaces have adopted peer-to-peer approaches in

which all nodes cooperate to provide the tuple space service. This approach is

particularly attractive given the intrinsic availability and scalability of peer-to-peer

solutions. Examples of peer-to-peer implementations include PeerSpaces [Busi et

al. 2003], which is developed using the JXTA peer-to-peer middleware

[jxta.dev.java.net], LIME and TOTA (the latter two systems feature in Chapter 19).

Case study: JavaSpaces • JavaSpaces is a tool for tuple space communication

developed by Sun [java.sun.com X, [java.sun.com VI]. More specifically, Sun provides

the specification of a JavaSpaces service, and third-party developers are then free to

offer implementations of JavaSpaces (significant implementations include GigaSpaces

[www.gigaspaces.com] and Blitz [www.dancres.org]). The tool is strongly dependent

on Jini (Sun’s discovery service, discussed further in Section 19.2.1), as will become

apparent below. The Jini Technology Starter Kit also includes an implementation of

JavaSpaces, referred to as Outrigger.

The goals of the JavaSpaces technology are:

• to offer a platform that simplifies the design of distributed applications and

services;

• to be simple and minimal in terms of the number and size of associated classes and

to have a small footprint to allow the code to run on resource-limited devices (such

as smart phones);

• to enable replicated implementations of the specification (although in practice

most implementations are centralized).

Figure 6.23 The JavaSpaces API

Operation Effect

Lease write(Entry e, Transaction txn, long lease) Places an entry into a particular

JavaSpace

Entry read(Entry tmpl, Transaction txn, long timeout) Returns a copy of an entry matching

a specified template

Entry readIfExists(Entry tmpl, Transaction txn, long timeout) As above, but not blocking

Entry take(Entry tmpl, Transaction txn, long timeout) Retrieves (and removes) an entry

matching a specified template

Entry takeIfExists(Entry tmpl, Transaction txn, long timeout) As above, but not blocking

EventRegistration notify(Entry tmpl, Transaction txn,

RemoteEventListener listen, long lease,

MarshalledObject handback)

Notifies a process if a tuple matching

a specified template is written to a

JavaSpace288 CHAPTER 6 INDIRECT COMMUNICATION

Programming with JavaSpaces: JavaSpaces allows the programmer to create any number

of instances of a space, where a space is a shared, persistent repository of objects (thus

offering an object space in the terminology introduced above). More specifically, an

item in a JavaSpace is referred to as an entry: a group of objects contained in a class that

implements net.jini.core.entry.Entry. Note that with entries containing objects (rather

than tuples), it is possible to associate arbitrary behaviour with entries, thus significantly

increasing the expressive power of the approach.

The operations defined on JavaSpaces are summarized in Figure 6.23 (showing

the full signatures of each of the operations) and can be described as follows:

• A process can place an entry into a JavaSpace instance with the write method. As

with Jini, an entry can have an associated lease (see Section 5.4.3), which is the

time for which access is granted to the associated objects. This can be forever

(Lease.FOREVER) or can be a numerical value specified in milliseconds. After

this period, the entry is destroyed. The write operation can also be used in the

context of a transaction, as discussed below (a value of null indicates that this is

not a transactional operation). The write operation returns a Lease value

representing the lease granted by the JavaSpace (which may be less than the time

requested).

• A process can access an entry in a JavaSpace with either the read or take

operation; read returns a copy of a matching entry and take removes a matching

entry from the JavaSpace (as in the general programming model presented above).

The matching requirements are specified by a template, which is of type entry.

Particular fields in the template may be set to specific values and others can be left

unspecified. A match is then defined as an entry that is of the same class as the

template (or a valid subclass) and where there is an exact match for the set of

specified values. As with write, read and take can be carried out in the context of

a specified transaction (discussed below). The two operations are also blocking;

the final parameter specifies a timeout representing the maximum length of time

that a particular process or thread will block, for example to deal with the failure

of a process supplying a given entry. The readIfExists and takeIfExists operations

Figure 6.24 Java class AlarmTupleJS

import net.jini.core.entry.\*;

public class AlarmTupleJS implements Entry {

public String alarmType;

public AlarmTupleJS() {

}

public AlarmTupleJS(String alarmType) {

this.alarmType = alarmType;

}

}SECTION 6.5 SHARED MEMORY APPROACHES 289

are equivalent to read and take, respectively, but these operations will return a

matching entry if one exists; otherwise, they will return null.

• The notify operation uses Jini distributed event notification, mentioned in Section

6.3 to register an interest in a given event – in this case, the arrival of entries

matching a given template. This registration is governed by a lease, that is, the

length of time the registration should persist in the JavaSpace. Notification is via

a specified RemoteEventListener interface. Once again, this operation can be

carried out in the context of a specified transaction.

As mentioned throughout the discussion above, operations in JavaSpaces can take place

in the context of a transaction, ensuring that either all or none of the operations will be

executed. Transactions are distributed entities and can span multiple JavaSpaces and

multiple participating processes. Discussion of the general concept of transactions is

deferred until Chapter 16.

A simple example: We conclude this examination of JavaSpaces by presenting an

example, the intelligent fire alarm example first introduced in Section 6.2.3 and revisited

in Section 6.4.3. In this example, there is a need to disseminate an emergency message

to all recipients when a fire event is detected.

We start by defining an entry object of type AlarmTupleJS, as shown in Figure

6.24. This is relatively straightforward and shows the creation of a new entry with one

field, the alarmType. The associated fire alarm code is shown in Figure 6.25. The first

step in raising an alarm is to gain access to an appropriate instance of a JavaSpace (called

"AlarmSpace"), which we assume is already created. Most implementations of

JavaSpaces provide utility functions for this and, for simplicity, this is what we show in

this code, using a utility class SpaceAccessor and method findSpace as provided in

GigaSpaces (for convenience, a copy of this class is provided on the companion web site

for the book [www.cdk5.net]). An entry is then created as an instance of the previously

defined AlarmTupleJS. This entry has only one field, a string called alarmType, and this

Figure 6.25 Java class FireAlarmJS

import net.jini.space.JavaSpace;

public class FireAlarmJS {

public void raise() {

try {

JavaSpace space = SpaceAccessor.findSpace("AlarmSpace");

AlarmTupleJS tuple = new AlarmTupleJS("Fire!");

space.write(tuple, null, 60\*60\*1000);

catch (Exception e) {

}

}

}290 CHAPTER 6 INDIRECT COMMUNICATION

is set to "Fire!"”. Finally, this entry is placed into the JavaSpace using the write method,

where it will remain for one hour. This code can then be called using the following:

FireAlarmJS alarm = new FireAlarmJS();

alarm.raise();

The corresponding code for the consumer end is shown in Figure 6.26. Access to the

appropriate JavaSpace is obtained in the same manner. Following this, a template is

created, the single field is set to "Fire!", and an associated read method is invoked. Note

that by setting the field to "Fire!", we ensure that only entries with this type and this

value will be returned (leaving the field blank would make any entry of type

AlarmTupleJS a valid match). This is called as follows in a consumer:

FireAlarmConsumerJS alarmCall = new FireAlarmConsumerJS();

String msg = alarmCall.await();

System.out.println("Alarm received: " + msg);

This simple example illustrates how easy it is to write multiparty applications using

JavaSpaces that are both time- and space-uncoupled.

6.6 Summary

This chapter has examined indirect communication in detail, complementing the study

of remote invocation paradigms in the previous chapter. We defined indirect

communication in terms of communication through an intermediary, with a resultant

uncoupling between producers and consumers of messages. This leads to interesting

properties, particularly in terms of dealing with change and establishing fault-tolerant

strategies.

Figure 6.26 Java class FireAlarmReceiverJS

import net.jini.space.JavaSpace;

public class FireAlarmConsumerJS {

public String await() {

try {

JavaSpace space = SpaceAccessor.findSpace();

AlarmTupleJS template = new AlarmTupleJS("Fire!");

AlarmTupleJS recvd = (AlarmTupleJS) space.read(template, null,

Long.MAX\_VALUE);

return recvd.alarmType;

}

catch (Exception e) {

return null;

}

}

}SECTION 6.6 SUMMARY 291

We have considered five styles of indirect communication in this chapter:

• group communication;

• publish-subscribe systems;

• message queues;

• distributed shared memory;

• tuple spaces.

The discussion has emphasized their commonalities in terms of all supporting indirect

communication through forms of intermediary including groups, channels or topics,

queues, shared memory or tuple spaces. Content-based publish-subscribe systems

communicate through the publish-subscribe system as a whole, with subscriptions

effectively defining logical channels managed by content-based routing.

As well as focusing on the commonalities, it is instructive to consider the key

differences between the various approaches. We start by reconsidering the level of space

and time uncoupling, picking up on the discussion in Section 6.1. All the techniques

considered in this chapter exhibit space uncoupling in that messages are directed to an

intermediary and not to any specific recipient or recipients. The position with respect to

time uncoupling is more subtle and dependent on the level of persistency in the

paradigm. Message queues, distributed shared memory and tuple spaces all exhibit time

uncoupling. The other paradigms may, depending on the implementation. For example,

in group communication, it is possible in some implementations for a receiver to join a

group at an arbitrary point in time and to be brought up-to-date with respect to previous

message exchanges (this is an optional feature in JGroups, for example, selected by

constructing an appropriate protocol stack). Many publish-subscribe systems do not

support persistency of events and hence are not time-uncoupled, but there are

exceptions. JMS, for example, does support persistent events, in keeping with its

integration of publish-subscribe and message queues.

The next observation is that the initial three techniques (groups, publish-subscribe

and message queues) offer a programming model that emphasizes communication

(through messages or events), whereas distributed shared memory and tuple spaces offer

a more state-based abstraction. This is a fundamental difference and one that has

significant repercussions in terms of scalability; in general terms, the communicationbased abstractions have the potential to scale to very large scale systems with

appropriate routing infrastructure (although this is not the case for group communication

because of the need to maintain group membership, as discussed in Section 6.2.2). In

contrast, the two state-based approaches have limitations with respect to scaling. This

stems from the need to maintain consistent views of the shared state, for example

between multiple readers and writers of shared memory. The situation with tuple spaces

is a bit more subtle given the immutable nature of tuples. The key problem rests with

implementing the destructive read operation, take, in a large-scale system; it is an

interesting observation that without this operation, tuple spaces look very much like

publish-subscribe systems (and hence are potentially highly scalable).

Most of the above systems also offer one-to-many styles of communication, that

is, multicast in terms of the communication-based services and global access to shared

values in the state-based abstractions. The exceptions are message queuing, which is

fundamentally point-to-point (and hence often offered in combination with publish292 CHAPTER 6 INDIRECT COMMUNICATION

subscribe systems in commercial middleware), tuple spaces, which can be either one-tomany or point-to-point depending on whether receiving processes use the read or take

operations, respectively.

There are also differences in intent in the various systems. Group communication

is mainly designed to support reliable distributed systems, and hence the emphasis is on

providing algorithmic support for reliability and ordering of message delivery.

Interestingly, the algorithms to ensure reliability and ordering (especially the latter) can

have a significant negative effect on scalability for similar reasons to maintaining

consistent views of shared state. Publish-subscribe systems have largely been targeted

at information dissemination (for example, in financial systems) and for Enterprise

Application Integration. Finally, the shared memory approaches have generally been

applied in parallel and distributed processing, including in the Grid community

(although tuple spaces have been used effectively across a variety of application

domains). Both publish-subscribe systems and tuple space communication have found

favour in mobile and ubiquitous computing due to their support for volatile

environments (as discussed in Chapter 19).

One other key issue associated with the five schemes is that both content-based

publish-subscribe and tuple spaces offer a form of associative addressing based on

content, allowing pattern matching between subscriptions and events or templates

against tuples, respectively. The other approaches do not.

This discussion is summarized in Figure 6.27.

Figure 6.27 Summary of indirect communication styles

Groups Publishsubscribe systems

Message queues DSM Tuple spaces

Spaceuncoupled

Yes Yes Yes Yes Yes

Time- uncoupled Possible Possible Yes Yes Yes

Style of service Communicationbased

Communicationbased

Communicationbased

State-based State-based

Communication

pattern

1-to-many 1-to-many 1-to-1 1-to-many 1-1 or 1-to-many

Main intent Reliable

distributed

computing

Information

dissemination or

EAI; mobile and

ubiquitous

systems

Information

dissemination or

EAI;

commercial

transaction

processing

Parallel and

distributed

computation

Parallel and

distributed

computation;

mobile and

ubiquitous

systems

Scalability Limited Possible Possible Limited Limited

Associative No Content-based

publish-subscribe

only

No No YesEXERCISES 293

We have not considered issues related to quality of service in this analysis. Many

message queue systems do offer intrinsic support for reliability in the form of

transactions. More generally, however, quality of service remains a key challenge for

indirect communication paradigms. Indeed, space and time uncoupling by their very

nature make it difficult to reason about end-to-end properties of the system, such as realtime behaviour or security, and hence this is an important area for further research.

EXERCISES

6.1 Indirect communication avoids direct coupling and hence inherits interesting properties.

Two key properties of this scheme are space and time uncoupling. Perform a

comparative study between space and time uncoupling. page 248

6.2 Section 6.1 states that message passing is both time- and space-coupled – that is,

messages are both directed towards a particular entity and require the receiver to be

present at the time of the message send. Consider the case, though, where messages are

directed towards a name rather than an address and this name is resolved using DNS.

Does such a system exhibit the same level of indirection? page 247, Section 13.2.3

6.3 Section 6.1 refers to systems that are space-coupled but time- uncoupled – that is,

messages are directed towards a given receiver (or receivers), but that receiver can have

a lifetime independent from the sender’s. Can you construct a communication paradigm

with these properties? For example, does email fall into this category? page 247

6.4 As a second example, consider the communication paradigm referred to as queued RPC,

as introduced in Rover [Joseph et al. 1997]. Rover is a toolkit to support distributed

systems programming in mobile environments where participants in communication

may become disconnected for periods of time. The system offers the RPC paradigm and

hence calls are directed towards a given server (clearly space-coupled). The calls,

though, are routed through an intermediary, a queue at the sending side, and are

maintained in the queue until the receiver is available. To what extent is this timeuncoupled? Hint: consider the almost philosophical question of whether a recipient that

is temporarily unavailable exists at that point in time. page 247, Chapter 19

6.5 Group communication is an important building block for reliable distributed systems.

Can you identify some key areas of application? page 249

6.6 Electra is a CORBA-compliant system. Explain how Electra works in ‘transparent

mode’ and in ‘non-transparent mode’. page 250

6.7 Consider the FireAlarm example as written using JGroups (Section 6.2.3). Suppose this

was generalized to support a variety of alarm types, such as fire, flood, intrusion and so

on. What are the requirements of this application in terms of reliability and ordering?

pages 246, 256294 CHAPTER 6 INDIRECT COMMUNICATION

6.8 Suggest a design for a notification mailbox service that is intended to store notifications

on behalf of multiple subscribers, allowing subscribers to specify when they require

notifications to be delivered. Explain how subscribers that are not always active can

make use of the service you describe. How will the service deal with subscribers that

crash while they have delivery turned on? page 261

6.9 In publish-subscribe systems, explain how channel-based approaches can trivially be

implemented using a group communication service? Why is this a less optimal strategy

for implementing a content-based approach? page 261

6.10 Using the filtering-based routing algorithm in Figure 6.11 as a starting point, develop an

alternative algorithm that illustrates how the use of advertisements can result in

significant optimization in terms of message traffic generated. page 267

6.11 Message queues are an important category of indirect communication systems. Why are

they also referred to as Message-Oriented Middleware? page 270

6.12 Building on your answer to Exercise 6.11, discuss two possible implementations of

EN(e) and SN(s). Why must the intersection of EN(e) and SN(s) be non-null for a given

e that matches s (the intersection rule)? Does this apply in your possible

implementations? page 268

6.13 Explain how the loose coupling inherent in message queues can aid with Enterprise

Application Integration. As in Exercise 6.1, consider to what extent this can be traced to

time uncoupling, space uncoupling or a combination of both. page 270

6.14 Consider the version of the FireAlarm program written in JMS (Section 6.4.3). How is

connecting to a named channel in JGroups achieved here?

page 277

6.15 Perform a comparative study between message passing and DSM.

page 280

6.16 What are the goals of JavaSpaces technology?

page 287

6.17 Assuming a DSM system is implemented in middleware without any hardware support

and in a platform-neutral manner, how would you deal with the problem of differing data

representations on heterogeneous computers? Does your solution extend to pointers?

page 278

6.18 How would you implement the equivalent of a remote procedure call using a tuple

space? What are the advantages and disadvantages of implementing a remote procedure

call–style interaction in this way? page 281

6.19 What are the basic operations defined on JavaSpaces? page 288, 289

6.20 Implement a replicated tuple space using the algorithm of Xu and Liskov [1989].

Explain how this algorithm uses the semantics of tuple space operations to optimize the

replication strategy. page 285295

7

OPERATING SYSTEM SUPPORT

7.1 Introduction

7.2 The operating system layer

7.3 Protection

7.4 Processes and threads

7.5 Communication and invocation

7.6 Operating system architecture

7.7 Virtualization at the operating system level

7.8 Summary

This chapter describes how middleware is supported by the operating system facilities at

the nodes of a distributed system. The operating system facilitates the encapsulation and

protection of resources inside servers and it supports the mechanisms required to access

these resources, including communication and scheduling.

An important theme of the chapter is the role of the system kernel. The chapter aims

to give the reader an understanding of the advantages and disadvantages of splitting

functionality between protection domains – in particular, of splitting functionality between

kernel- and user-level code. The trade-offs between kernel-level facilities and user-level

facilities are discussed, including the tension between efficiency and robustness.

The chapter examines the design and implementation of multi-threaded processing

and communication facilities. It goes on to explore the main kernel architectures that have

been devised and looks at the important role that virtualization is playing in operating

system architecture.296 CHAPTER 7 OPERATING SYSTEM SUPPORT

7.1 Introduction

Chapter 2 introduced the chief software layers in a distributed system. We have learned

that an important aspect of distributed systems is resource sharing. Client applications

invoke operations on resources that are often on another node or at least in another

process. Applications (in the form of clients) and services (in the form of resource

managers) use the middleware layer for their interactions. Middleware enables remote

communication between objects or processes at the nodes of a distributed system.

Chapter 5 explained the main types of remote invocation found in middleware, such as

Java RMI and CORBA, with Chapter 6 exploring alternative indirect styles of

communication. In this chapter we shall focus on support for such remote

communication, without real-time guarantees. (Chapter 20 examines support for

multimedia communication, which is real-time and stream-oriented.)

Below the middleware layer is the operating system (OS) layer, which is the

subject of this chapter. Here we examine the relationship between the two, and in

particular how well the requirements of middleware can be met by the operating system.

Those requirements include efficient and robust access to physical resources, and the

flexibility to implement a variety of resource-management policies.

The task of any operating system is to provide problem-oriented abstractions of

the underlying physical resources – the processors, memory, networks, and storage

media. An operating system such as UNIX (and its variants, such as Linux and Mac OS

X) or Windows (and its variants, such as XP, Vista and Windows 7) provides the

programmer with, for example, files rather than disk blocks, and with sockets rather than

raw network access. It takes over the physical resources on a single node and manages

them to present these resource abstractions through the system-call interface.

Before we begin our detailed coverage of the operating system’s middleware

support role, it is useful to gain some historical perspective by examining two operating

system concepts that have come about during the development of distributed systems:

network operating systems and distributed operating systems. Definitions vary, but the

concepts behind them are something like the following.

Both UNIX and Windows are examples of network operating systems. They have

a networking capability built into them and so can be used to access remote resources.

Access is network-transparent for some – not all – types of resource. For example,

through a distributed file system such as NFS, users have network-transparent access to

files. That is, many of the files that users access are stored remotely, on a server, and this

is largely transparent to their applications.

But the defining characteristic is that the nodes running a network operating

system retain autonomy in managing their own processing resources. In other words,

there are multiple system images, one per node. With a network operating system, a user

can remotely log into another computer, using ssh, for example, and run processes there.

However, while the operating system manages the processes running at its own node, it

does not manage processes across the nodes.

By contrast, one could envisage an operating system in which users are never

concerned with where their programs run, or the location of any resources. There is a

single system image. The operating system has control over all the nodes in the system,

and it transparently locates new processes at whatever node suits its scheduling policies.SECTION 7.2 THE OPERATING SYSTEM LAYER 297

For example, it could create a new process at the least-loaded node in the system, to

prevent individual nodes becoming unfairly overloaded.

An operating system that produces a single system image like this for all the

resources in a distributed system is called a distributed operating system [Tanenbaum

and van Renesse 1985].

Middleware and network operating systems • In fact, there are no distributed operating

systems in general use, only network operating systems such as UNIX, Mac OS and

Windows. This is likely to remain the case, for two main reasons. The first is that users

have much invested in their application software, which often meets their current

problem-solving needs; they will not adopt a new operating system that will not run their

applications, whatever efficiency advantages it offers. Attempts have been made to

emulate UNIX and other operating system kernels on top of new kernels, but the

emulations’ performance has not been satisfactory. Anyway, keeping emulations of all

the major operating systems up-to-date as they evolve would be a huge undertaking.

The second reason against the adoption of distributed operating systems is that

users tend to prefer to have a degree of autonomy for their machines, even in a closely

knit organization. This is particularly so because of performance [Douglis and

Ousterhout 1991]. For example, Jones needs good interactive responsiveness while she

writes her documents and would resent it if Smith’s programs were slowing her down.

The combination of middleware and network operating systems provides an

acceptable balance between the requirement for autonomy on the one hand and networktransparent resource access on the other. The network operating system enables users to

run their favourite word processors and other standalone applications. Middleware

enables them to take advantage of services that become available in their distributed

system.

The next section explains the function of the operating system layer. Section 7.3

examines low-level mechanisms for resource protection, which we need to understand

in order to appreciate the relationship between processes and threads, and the role of the

kernel itself. Section 7.4 goes on to examine the process, address space and thread

abstractions. Here the main topics are concurrency, local resource management and

protection, and scheduling. Section 7.5 then covers communication as part of invocation

mechanisms. Section 7.6 discusses the different types of operating system architecture,

including the so-called monolithic and microkernel designs. The reader can find case

studies of the Mach kernel and the Amoeba, Chorus and Clouds operating systems at

www.cdk5.net/oss. The chapter concludes by examining the role that virtualization is

playing in the design of operating systems, featuring a case study of the Xen approach

to virtualization (Section 7.7).

7.2 The operating system layer

Users will only be satisfied if their middleware–OS combination has good performance.

Middleware runs on a variety of OS–hardware combinations (platforms) at the nodes of

a distributed system. The OS running at a node – a kernel and associated user-level

services such as communication libraries – provides its own flavour of abstractions of

local hardware resources for processing, storage and communication. Middleware298 CHAPTER 7 OPERATING SYSTEM SUPPORT

utilizes a combination of these local resources to implement its mechanisms for remote

invocations between objects or processes at the nodes.

Figure 7.1 shows how the operating system layer at each of two nodes supports a

common middleware layer in providing a distributed infrastructure for applications and

services.

Our goal in this chapter is to examine the impact of particular OS mechanisms on

middleware’s ability to deliver distributed resource sharing to users. Kernels and the

client and server processes that execute upon them are the chief architectural

components that concern us. Kernels and server processes are the components that

manage resources and present clients with an interface to the resources. As such, we

require at least the following of them:

Encapsulation: They should provide a useful service interface to their resources –

that is, a set of operations that meet their clients’ needs. Details such as management

of memory and devices used to implement resources should be hidden from clients.

Protection: Resources require protection from illegitimate accesses – for example,

files are protected from being read by users without read permissions, and device

registers are protected from application processes.

Concurrent processing: Clients may share resources and access them concurrently.

Resource managers are responsible for achieving concurrency transparency.

Clients access resources by making, for example, remote method invocations to a server

object, or system calls to a kernel. We call a means of accessing an encapsulated

resource an invocation mechanism, however it is implemented. A combination of

libraries, kernels and servers may be called upon to perform the following invocationrelated tasks:

Communication: Operation parameters and results have to be passed to and from

resource managers, over a network or within a computer.

Scheduling: When an operation is invoked, its processing must be scheduled within

the kernel or server.

Applications, services

Computer &

Figure 7.1 System layers

Platform

Middleware

OS: kernel,

libraries &

servers

network hardware

OS1

Computer &

network hardware

Node 1 Node 2

Processes, threads,

communication, ...

OS2

Processes, threads,

communication, ...SECTION 7.2 THE OPERATING SYSTEM LAYER 299

Figure 7.2 shows the core OS functionality that we shall be concerned with: process and

thread management, memory management and communication between processes on

the same computer (horizontal divisions in the figure denote dependencies). The kernel

supplies much of this functionality – all of it in the case of some operating systems.

OS software is designed to be portable between computer architectures where

possible. This means that the majority of it is coded in a high-level language such as C,

C++ or Modula-3, and that its facilities are layered so that machine-dependent

components are reduced to a minimal bottom layer. Some kernels can execute on

shared-memory multiprocessors, which are described in the box below.

Figure 7.2 Core OS functionality

Communication

manager

Thread manager Memory manager

Supervisor

Process manager

Shared-memory multiprocessors • Shared-memory multiprocessor computers are

equipped with several processors that share one or more modules of memory (RAM).

The processors may also have their own private memory. Multiprocessor computers

can be constructed in a variety of forms [Stone 1993]. The simplest and least

expensive multiprocessors are constructed by incorporating a circuit board holding a

few (2–8) processors in a personal computer.

In the common symmetric processing architecture, each processor executes the

same kernel and the kernels play largely equivalent roles in managing the hardware

resources. The kernels share key data structures, such as the queue of runnable

threads, but some of their working data is private. Each processor can execute a

thread simultaneously, accessing data in the shared memory, which may be private

(hardware-protected) or shared with other threads.

Multiprocessors can be used for many high-performance computing tasks. In

distributed systems, they are particularly useful for the implementation of highperformance servers because the server can run a single program with several threads

that handle several requests from clients simultaneously – for example, providing

access to a shared database (see Section 7.4)300 CHAPTER 7 OPERATING SYSTEM SUPPORT

The core OS components and their responsibilities are:

Process manager: Creation of and operations upon processes. A process is a unit of

resource management, including an address space and one or more threads.

Thread manager: Thread creation, synchronization and scheduling. Threads are

schedulable activities attached to processes and are fully described in Section 7.4.

Communication manager: Communication between threads attached to different

processes on the same computer. Some kernels also support communication between

threads in remote processes. Other kernels have no notion of other computers built

into them, and an additional service is required for external communication. Section

7.5 discusses the communication design.

Memory manager: Management of physical and virtual memory. Section 7.4 and

Section 7.5 describe the utilization of memory management techniques for efficient

data copying and sharing.

Supervisor: Dispatching of interrupts, system call traps and other exceptions; control

of memory management unit and hardware caches; processor and floating-point unit

register manipulations. This is known as the Hardware Abstraction Layer in

Windows. The reader is referred to Bacon [2002] and Tanenbaum [2007] for a fuller

description of the computer-dependent aspects of the kernel.

7.3 Protection

We said above that resources require protection from illegitimate accesses. However,

threats to a system’s integrity do not come only from maliciously contrived code.

Benign code that contains a bug or that has unanticipated behaviour may cause part of

the rest of the system to behave incorrectly.

To understand what we mean by an ‘illegitimate access’ to a resource, consider a

file. Let us suppose, for the sake of explanation, that open files have only two operations,

read and write. Protecting the file consists of two sub-problems. The first is to ensure

that each of the file’s two operations can be performed only by clients with the right to

perform it. For example, Smith, who owns the file, has read and write rights to it. Jones

may only perform the read operation. An illegitimate access here would be if Jones

somehow managed to perform a write operation on the file. A complete solution to this

resource-protection sub-problem in a distributed system requires cryptographic

techniques, and we defer it to Chapter 11.

The other type of illegitimate access, which we address here, is where a

misbehaving client sidesteps the operations that a resource exports. In our example, this

would be if Smith or Jones somehow managed to execute an operation that was neither

read nor write. Suppose, for example, that Smith managed to access the file pointer

variable directly. She could then construct a setFilePointerRandomly operation, that sets

the file pointer to a random number. Of course, this is a meaningless operation that

would upset normal use of the file.SECTION 7.3 PROTECTION 301

We can protect resources from illegitimate invocations such as

setFilePointerRandomly. One way is to use a type-safe programming language, such as

Sing#, an extension of C# used in the Singularity project [Hunt et al. 2007], or Modula-

3. In type-safe languages, no module may access a target module unless it has a

reference to it – it cannot make up a pointer to it, as would be possible in C or C++ – and

it may only use its reference to the target module to perform the invocations (method

calls or procedure calls) that the programmer of the target made available to it. It may

not, in other words, arbitrarily change the target’s variables. By contrast, in C++ the

programmer may cast a pointer however she likes, and thus perform non-type-safe

invocations.

We can also employ hardware support to protect modules from one another at the

level of individual invocations, regardless of the language in which they are written. To

operate this scheme on a general-purpose computer, we require a kernel.

Kernels and protection • The kernel is a program that is distinguished by the facts that

it remains loaded from system initialization and its code is executed with complete

access privileges for the physical resources on its host computer. In particular, it can

control the memory management unit and set the processor registers so that no other

code may access the machine’s physical resources except in acceptable ways.

Most processors have a hardware mode register whose setting determines whether

privileged instructions can be executed, such as those used to determine which

protection tables are currently employed by the memory management unit. A kernel

process executes with the processor in supervisor (privileged) mode; the kernel arranges

that other processes execute in user (unprivileged) mode.

The kernel also sets up address spaces to protect itself and other processes from

the accesses of an aberrant process, and to provide processes with their required virtual

memory layout. An address space is a collection of ranges of virtual memory locations,

in each of which a specified combination of memory access rights applies, such as readonly or read-write. A process cannot access memory outside its address space. The terms

user process or user-level process are normally used to describe one that executes in

user mode and has a user-level address space (that is, one with restricted memory access

rights compared with the kernel’s address space).

When a process executes application code, it executes in a distinct user-level

address space for that application; when the same process executes kernel code, it

executes in the kernel’s address space. The process can safely transfer from a user-level

address space to the kernel’s address space via an exception such as an interrupt or a

system call trap – the invocation mechanism for resources managed by the kernel. A

system call trap is implemented by a machine-level TRAP instruction, which puts the

processor into supervisor mode and switches to the kernel address space. When the

TRAP instruction is executed, as with any type of exception, the hardware forces the

processor to execute a kernel-supplied handler function, in order that no process may

gain illicit control of the hardware.

Programs pay a price for protection. Switching between address spaces may take

many processor cycles, and a system call trap is a more expensive operation than a

simple procedure or method call. We shall see in Section 7.5.1 how these penalties factor

into invocation costs.302 CHAPTER 7 OPERATING SYSTEM SUPPORT

7.4 Processes and threads

The traditional operating system notion of a process that executes a single activity was

found in the 1980s to be unequal to the requirements of distributed systems – and also

to those of more sophisticated single-computer applications that require internal

concurrency. The problem, as we shall show, is that the traditional process makes

sharing between related activities awkward and expensive.

The solution reached was to enhance the notion of a process so that it could be

associated with multiple activities. Nowadays, a process consists of an execution

environment together with one or more threads. A thread is the operating system

abstraction of an activity (the term derives from the phrase ‘thread of execution’). An

execution environment is the unit of resource management: a collection of local kernelmanaged resources to which its threads have access. An execution environment

primarily consists of:

• an address space;

• thread synchronization and communication resources such as semaphores and

communication interfaces (for example, sockets);

• higher-level resources such as open files and windows.

Execution environments are normally expensive to create and manage, but several

threads can share them – that is, they can share all resources accessible within them. In

other words, an execution environment represents the protection domain in which its

threads execute.

Threads can be created and destroyed dynamically, as needed. The central aim of

having multiple threads of execution is to maximize the degree of concurrent execution

between operations, thus enabling the overlap of computation with input and output, and

enabling concurrent processing on multiprocessors. This can be particularly helpful

within servers, where concurrent processing of clients’ requests can reduce the tendency

for servers to become bottlenecks. For example, one thread can process a client’s request

while a second thread servicing another request waits for a disk access to complete.

An execution environment provides protection from threads outside it, so that the

data and other resources contained in it are by default inaccessible to threads residing in

An analogy for threads and processes • The following memorable, if slightly unsavoury, way to think of the concepts of threads and execution environments was published on the comp.os.mach USENET group and is by Chris Lloyd. An execution

environment consists of a stoppered jar and the air and food within it. Initially, there

is one fly – a thread – in the jar. This fly can produce other flies and kill them, as can

its progeny. Any fly can consume any resource (air or food) in the jar. Flies can be

programmed to queue up in an orderly manner to consume resources. If they lack this

discipline, they might bump into one another within the jar – that is, collide and produce unpredictable results when attempting to consume the same resources in an unconstrained manner. Flies can communicate with (send messages to) flies in other

jars, but none may escape from the jar, and no fly from outside may enter it. In this

view, originally a UNIX process was a single jar with a single sterile fly within it.SECTION 7.4 PROCESSES AND THREADS 303

other execution environments. But certain kernels allow the controlled sharing of

resources such as physical memory between execution environments residing at the

same computer.

As many older operating systems allow only one thread per process, we shall

sometimes use the term multi-threaded process for emphasis. Confusingly, in some

programming models and operating system designs the term ‘process’ means what we

have called a thread. The reader may encounter in the literature the terms heavyweight

process, where an execution environment is taken to be included, and lightweight

process, where it is not. See the box on the preceding page for an analogy describing

threads and execution environments.

7.4.1 Address spaces

An address space, introduced in the previous section, is a unit of management of a

process’s virtual memory. It is large (typically up to 232 bytes, and sometimes up to 264

bytes) and consists of one or more regions, separated by inaccessible areas of virtual

memory. A region (Figure 7.3) is an area of contiguous virtual memory that is accessible

by the threads of the owning process. Regions do not overlap. Note that we distinguish

between the regions and their contents. Each region is specified by the following

properties:

• its extent (lowest virtual address and size);

• read/write/execute permissions for the process’s threads;

• whether it can be grown upwards or downwards.

Figure 7.3 Address space

Stack

Text

Heap

Auxiliary

regions

0

2N304 CHAPTER 7 OPERATING SYSTEM SUPPORT

Note that this model is page-oriented rather than segment-oriented. Regions, unlike

segments, would eventually overlap if they were extended in size. Gaps are left between

regions to allow for growth. This representation of an address space as a sparse set of

disjoint regions is a generalization of the UNIX address space, which has three regions:

a fixed, unmodifiable text region containing program code; a heap, part of which is

initialized by values stored in the program’s binary file, and which is extensible towards

higher virtual addresses; and a stack, which is extensible towards lower virtual

addresses.

The provision of an indefinite number of regions is motivated by several factors.

One of these is the need to support a separate stack for each thread. Allocating a separate

stack region to each thread makes it possible to detect attempts to exceed the stack limits

and to control each stack’s growth. Unallocated virtual memory lies beyond each stack

region, and attempts to access this will cause an exception (a page fault). The alternative

is to allocate stacks for threads on the heap, but then it is difficult to detect when a thread

has exceeded its stack limit.

Another motivation is to enable files in general – not just the text and data sections

of binary files – to be mapped into the address space. A mapped file is one that is

accessed as an array of bytes in memory. The virtual memory system ensures that

accesses made in memory are reflected in the underlying file storage. Section

CDK3-18.6 (in www.cdk5.net/oss/mach) describes how the Mach kernel extends the

abstraction of virtual memory so that regions can correspond to arbitrary ‘memory

objects’ and not just to files.

The need to share memory between processes, or between processes and the

kernel, is another factor leading to extra regions in the address space. A shared memory

region (or shared region for short) is one that is backed by the same physical memory

as one or more regions belonging to other address spaces. Processes therefore access

identical memory contents in the regions that are shared, while their non-shared regions

remain protected. The uses of shared regions include the following:

Libraries: Library code can be very large and would waste considerable memory if it

was loaded separately into every process that used it. Instead, a single copy of the

library code can be shared by being mapped as a region in the address spaces of

processes that require it.

Kernel: Often the kernel code and data are mapped into every address space at the

same location. When a process makes a system call or an exception occurs, there is

no need to switch to a new set of address mappings.

Data sharing and communication: Two processes, or a process and the kernel, might

need to share data in order to cooperate on some task. It can be considerably more

efficient for the data to be shared by being mapped as regions in both address spaces

than by being passed in messages between them. The use of region sharing for

communication is described in Section 7.5.SECTION 7.4 PROCESSES AND THREADS 305

7.4.2 Creation of a new process

The creation of a new process has traditionally been an indivisible operation provided

by the operating system. For example, the UNIX fork system call creates a process with

an execution environment copied from the caller (except for the return value from fork).

The UNIX exec system call transforms the calling process into one executing the code

of a named program.

For a distributed system, the design of the process-creation mechanism has to take

into account the utilization of multiple computers; consequently, the process-support

infrastructure is divided into separate system services.

The creation of a new process can be separated into two independent aspects:

• the choice of a target host, for example, the host may be chosen from among the

nodes in a cluster of computers acting as a compute server, as introduced in

Chapter 1;

• the creation of an execution environment (and an initial thread within it).

Choice of process host • The choice of the node at which the new process will reside –

the process allocation decision – is a matter of policy. In general, process allocation

policies range from always running new processes at their originator’s workstation to

sharing the processing load between a set of computers. Eager et al. [1986] distinguish

two policy categories for load sharing.

The transfer policy determines whether to situate a new process locally or

remotely. This may depend, for example, on whether the local node is lightly or heavily

loaded.

The location policy determines which node should host a new process selected for

transfer. This decision may depend on the relative loads of nodes, on their machine

architectures or on any specialized resources they may possess. The V system [Cheriton

1984] and Sprite [Douglis and Ousterhout 1991] both provide commands for users to

execute a program at a currently idle workstation (there are often many of these at any

given time) chosen by the operating system. In the Amoeba system [Tanenbaum et al.

1990], the run server chooses a host for each process from a shared pool of processors.

In all cases, the choice of target host is transparent to the programmer and the user.

Those programming for explicit parallelism or fault tolerance, however, may require a

means of specifying process location.

Process location policies may be static or adaptive. The former operate without

regard to the current state of the system, although they are designed according to the

system’s expected long-term characteristics. They are based on a mathematical analysis

aimed at optimizing a parameter such as the overall process throughput. They may be

deterministic (‘node A should always transfer processes to node B’) or probabilistic

(‘node A should transfer processes to any of nodes B–E at random’). Adaptive policies,

on the other hand, apply heuristics to make their allocation decisions, based on

unpredictable runtime factors such as a measure of the load on each node.

Load-sharing systems may be centralized, hierarchical or decentralized. In the

first case there is one load manager component, and in the second there are several,

organized in a tree structure. Load managers collect information about the nodes and use

it to allocate new processes to nodes. In hierarchical systems, managers make process

allocation decisions as far down the tree as possible, but managers may transfer306 CHAPTER 7 OPERATING SYSTEM SUPPORT

processes to one another, via a common ancestor, under certain load conditions. In a

decentralized load-sharing system, nodes exchange information with one another

directly to make allocation decisions. The Spawn system [Waldspurger et al. 1992], for

example, considers nodes to be ‘buyers’ and ‘sellers’ of computational resources and

arranges them in a (decentralized) ‘market economy’.

In sender-initiated load-sharing algorithms, the node that requires a new process

to be created is responsible for initiating the transfer decision. It typically initiates a

transfer when its own load crosses a threshold. By contrast, in receiver-initiated

algorithms, a node whose load is below a given threshold advertises its existence to other

nodes so that relatively loaded nodes can transfer work to it.

Migratory load-sharing systems can shift load at any time, not just when a new

process is created. They use a mechanism called process migration: the transfer of an

executing process from one node to another. Milojicic et al. [1999] provide a collection

of papers on process migration and other types of mobility. While several process

migration mechanisms have been constructed, they have not been widely deployed. This

is largely because of their expense and the tremendous difficulty of extracting the state

of a process that lies within the kernel, in order to move it to another node.

Eager et al. [1986] studied three approaches to load sharing and concluded that

simplicity is an important property of any load-sharing scheme. This is because

relatively high overheads – for example, state-collection overheads – can outweigh the

advantages of more complex schemes.

Creation of a new execution environment • Once the host computer has been selected, a

new process requires an execution environment consisting of an address space with

initialized contents (and perhaps other resources, such as default open files).

There are two approaches to defining and initializing the address space of a newly

created process. The first approach is used where the address space is of a statically

defined format. For example, it could contain just a program text region, heap region and

stack region. In this case, the address space regions are created from a list specifying

their extent. Address space regions are initialized from an executable file or filled with

zeros as appropriate.

Alternatively, the address space can be defined with respect to an existing

execution environment. In the case of UNIX fork semantics, for example, the newly

created child process physically shares the parent’s text region and has heap and stack

regions that are copies of the parent’s in extent (as well as in initial contents). This

scheme has been generalized so that each region of the parent process may be inherited

by (or omitted from) the child process. An inherited region may either be shared with or

logically copied from the parent’s region. When parent and child share a region, the page

frames (units of physical memory corresponding to virtual memory pages) belonging to

the parent’s region are mapped simultaneously into the corresponding child region.

Mach [Accetta et al. 1986] and Chorus [Rozier et al. 1988, 1990], for example,

apply an optimization called copy-on-write when an inherited region is copied from the

parent. The region is copied, but no physical copying takes place by default. The page

frames that make up the inherited region are shared between the two address spaces. A

page in the region is only physically copied when one or another process attempts to

modify it.SECTION 7.4 PROCESSES AND THREADS 307

Copy-on-write is a general technique – for example, it is also used in copying

large messages – so we take some time to explain its operation here. Let us follow

through an example of regions RA and RB, whose memory is shared copy-on-write

between two processes, A and B (Figure 7.4). For the sake of definiteness, let us assume

that process A set region RA to be copy-inherited by its child, process B, and that the

region RB was thus created in process B.

We assume, for the sake of simplicity, that the pages belonging to region A are

resident in memory. Initially, all page frames associated with the regions are shared

between the two processes’ page tables. The pages are initially write-protected at the

hardware level, even though they may belong to regions that are logically writable. If a

thread in either process attempts to modify the data, a hardware exception called a page

fault is taken. Let us say that process B attempted the write. The page fault handler

allocates a new frame for process B and copies the original frame’s data into it byte for

byte. The old frame number is replaced by the new frame number in one process’s page

table – it does not matter which – and the old frame number is left in the other page table.

The two corresponding pages in processes A and B are then each made writable once

more at the hardware level. After all of this has taken place, process B’s modifying

instruction is allowed to proceed.

Figure 7.4 Copy-on-write

a) Before write b) After write

Shared

frame

A’s page

table

B’s page

table

Process A’s address space Process B’s address space

Kernel

RA RB

RB copied

from RA308 CHAPTER 7 OPERATING SYSTEM SUPPORT

7.4.3 Threads

The next key aspect of a process to consider in more detail is its threads. This section

examines the advantages of enabling client and server processes to possess more than

one thread. It then discusses programming with threads, using Java threads as a case

study, and ends with alternative designs for implementing threads.

Consider the server shown in Figure 7.5 (we shall turn to the client shortly). The

server has a pool of one or more threads, each of which repeatedly removes a request

from a queue of received requests and processes it. We shall not concern ourselves for

the moment with how the requests are received and queued up for the threads. Also, for

the sake of simplicity, we assume that each thread applies the same procedure to process

the requests. Let us assume that each request takes, on average, 2 milliseconds of

processing plus 8 milliseconds of I/O (input/output) delay when the server reads from a

disk (there is no caching). Let us further assume for the moment that the server executes

at a single-processor computer.

Consider the maximum server throughput, measured in client requests handled per

second, for different numbers of threads. If a single thread has to perform all processing,

then the turnaround time for handling any request is on average 2 + 8 = 10 milliseconds,

so this server can handle 100 client requests per second. Any new request messages that

arrive while the server is handling a request are queued at the server port.

Now consider what happens if the server pool contains two threads. We assume

that threads are independently schedulable – that is, one thread can be scheduled when

another becomes blocked for I/O. Then thread number two can process a second request

while thread number one is blocked, and vice versa. This increases the server

throughput. Unfortunately, in our example, the threads may become blocked behind the

single disk drive. If all disk requests are serialized and take 8 milliseconds each, then the

maximum throughput is 1000/8 = 125 requests per second.

Suppose, now, that disk block caching is introduced. The server keeps the data that

it reads in buffers in its address space; a server thread that is asked to retrieve data first

examines the shared cache and avoids accessing the disk if it finds the data there. If a

75% hit rate is achieved, the mean I/O time per request reduces to (0.75×0 + 0.25×8) =

2 milliseconds, and the maximum theoretical throughput increases to 500 requests per

Figure 7.5 Client and server with threads

Server

N threads

Input-output

Client

Thread 2 makes

Thread 1

requests to server

generates

results

Requests

Receipt &

queuingSECTION 7.4 PROCESSES AND THREADS 309

second. But if the average processor time for a request has been increased to 2.5

milliseconds per request as a result of caching (it takes time to search for cached data on

every operation), then this figure cannot be reached. The server, limited by the

processor, can now handle at most 1000/2.5 = 400 requests per second.

The throughput can be increased by using a shared-memory multiprocessor to

ease the processor bottleneck. A multi-threaded process maps naturally onto a sharedmemory multiprocessor. The shared execution environment can be implemented in

shared memory, and the multiple threads can be scheduled to run on the multiple

processors. Consider now the case in which our example server executes at a

multiprocessor with two processors. Given that threads can be independently scheduled

to the different processors, then up to two threads can process requests in parallel. The

reader should check that two threads can process 444 requests per second and three or

more threads, bounded by the I/O time, can process 500 requests per second.

Architectures for multi-threaded servers • We have described how multi-threading

enables servers to maximize their throughput, measured as the number of requests

processed per second. To describe the various ways of mapping requests to threads

within a server we summarize the account by Schmidt [1998], who describes the

threading architectures of various implementations of the CORBA Object Request

Broker (ORB). ORBs process requests that arrive over a set of connected sockets. Their

threading architectures are relevant to many types of server, regardless of whether

CORBA is used.

Figure 7.5 shows one of the possible threading architectures, the worker pool

architecture. In its simplest form, the server creates a fixed pool of ‘worker’ threads to

process the requests when it starts up. The module marked ‘receipt and queuing’ in

Figure 7.5 is typically implemented by an ‘I/O’ thread, which receives requests from a

collection of sockets or ports and places them on a shared request queue for retrieval by

the workers.

There is sometimes a requirement to treat the requests with varying priorities. For

example, a corporate web server could prioritize request processing according to the

class of customer from which the request derives [Bhatti and Friedrich 1999]. We may

handle varying request priorities by introducing multiple queues into the worker pool

architecture, so that the worker threads scan the queues in the order of decreasing

priority. A disadvantage of this architecture is its inflexibility: as we saw with our

worked-out example, the number of worker threads in the pool may be too few to deal

adequately with the current rate of request arrival. Another disadvantage is the high level

of switching between the I/O and worker threads as they manipulate the shared queue.

In the thread-per-request architecture (Figure 7.6a) the I/O thread spawns a new

worker thread for each request, and that worker destroys itself when it has processed the

request against its designated remote object. This architecture has the advantage that the

threads do not contend for a shared queue, and throughput is potentially maximized

because the I/O thread can create as many workers as there are outstanding requests. Its

disadvantage is the overhead of the thread creation and destruction operations.

The thread-per-connection architecture (Figure 7.6b) associates a thread with

each connection. The server creates a new worker thread when a client makes a

connection and destroys the thread when the client closes the connection. In between,

the client may make many requests over the connection, targeted at one or more remote310 CHAPTER 7 OPERATING SYSTEM SUPPORT

objects. The thread-per-object architecture (Figure 7.6c) associates a thread with each

remote object. An I/O thread receives requests and queues them for the workers, but this

time there is a per-object queue.

In each of these last two architectures the server benefits from lower threadmanagement overheads compared with the thread-per-request architecture. Their

disadvantage is that clients may be delayed while a worker thread has several

outstanding requests but another thread has no work to perform.

Schmidt [1998] describes variations on these architectures as well as hybrids of

them, and discusses their advantages and disadvantages in more detail. Section 7.5

describes a different threading model in the context of invocations within a single

machine, in which client threads enter the server’s address space.

Threads within clients • Threads can be useful for clients as well as servers. Figure 7.5

also shows a client process with two threads. The first thread generates results to be

passed to a server by remote method invocation, but does not require a reply. Remote

method invocations typically block the caller, even when there is strictly no need to wait.

This client process can incorporate a second thread, which performs the remote method

invocations and blocks while the first thread is able to continue computing further

results. The first thread places its results in buffers, which are emptied by the second

thread. It is only blocked when all the buffers are full.

The case for multi-threaded clients is also evident in the example of web browsers.

Users experience substantial delays while pages are fetched; it is essential, therefore, for

browsers to handle multiple concurrent requests for web pages.

Threads versus multiple processes • We can see from the above examples the utility of

threads, which allow computation to be overlapped with I/O and, in the case of a

multiprocessor, with other computation. The reader may have noted, however, that the

same overlap could be achieved through the use of multiple single-threaded processes.

Why, then, should the multi-threaded process model be preferred? The answer is

twofold: threads are cheaper to create and manage than processes, and resource sharing

can be achieved more efficiently between threads than between processes because

threads share an execution environment.

Figure 7.7 shows some of the main state components that must be maintained for

execution environments and threads, respectively. An execution environment has an

address space, communication interfaces such as sockets, higher-level resources such as

open files and thread synchronization objects such as semaphores; it also lists the

Figure 7.6 Alternative server threading architectures (see also Figure 7.5)

a. Thread-per-request b. Thread-per-connection c. Thread-per-object

remote

workers

I/O remote I/O remote

per-connection threads per-object threads

objects objects objectsSECTION 7.4 PROCESSES AND THREADS 311

threads associated with it. A thread has a scheduling priority, an execution state (such as

BLOCKED or RUNNABLE), saved processor register values when the thread is

BLOCKED, and state concerning the thread’s software interrupt handling. A software

interrupt is an event that causes a thread to be interrupted (similar to the case of a

hardware interrupt). If the thread has assigned a handler procedure, control is transferred

to it. UNIX signals are examples of software interrupts.

The figure shows that an execution environment and the threads belonging to it

are both associated with pages belonging to the address space held in main memory, and

data and instructions held in hardware caches.

We can summarize a comparison of processes and threads as follows:

• Creating a new thread within an existing process is cheaper than creating a

process.

• More importantly, switching to a different thread within the same process is

cheaper than switching between threads belonging to different processes.

• Threads within a process may share data and other resources conveniently and

efficiently compared with separate processes.

• But, by the same token, threads within a process are not protected from one

another.

Consider the cost of creating a new thread in an existing execution environment. The

main tasks are to allocate a region for its stack and to provide initial values for the

processor registers and the thread’s execution state (it may initially be SUSPENDED or

RUNNABLE) and priority. Since the execution environment exists, only an identifier for

this has to be placed in the thread’s descriptor record (which contains data necessary to

manage the thread’s execution).

The overheads associated with creating a process are in general considerably

greater than those of creating a new thread. A new execution environment must first be

created, including address space tables. Anderson et al. [1991] quote a figure of about

11 milliseconds to create a new UNIX process, and about 1 millisecond to create a

thread on the same CVAX processor architecture running the Topaz kernel; in each case

the time measured includes the new entity simply calling a null procedure and then

exiting. These figures are given as a rough guide only.

Figure 7.7 State associated with execution environments and threads

Execution environment Thread

Address space tables Saved processor registers

Communication interfaces, open files Priority and execution state (such as

BLOCKED)

Semaphores, other synchronization

objects

Software interrupt handling information

List of thread identifiers Execution environment identifier

Pages of address space resident in memory; hardware cache entries312 CHAPTER 7 OPERATING SYSTEM SUPPORT

When the new entity performs some useful work rather than calling a null

procedure, there are also long-term costs, which are liable to be greater for a new process

than for a new thread within an existing process. In a kernel supporting virtual memory,

the new process will incur page faults as data and instructions are referenced for the first

time; hardware caches will initially contain no data values for the new process, and it

must acquire cache entries as it executes. In the case of thread creation, these long-term

overheads may also occur, but they are liable to be smaller. When the thread accesses

code and data that have recently been accessed by other threads within the process, it

automatically takes advantage of any hardware or main memory caching that has taken

place.

The second performance advantage of threads concerns switching between

threads – that is, running one thread instead of another at a given processor. This cost is

the most important, because it may be incurred many times in the lifetime of a thread.

Switching between threads sharing the same execution environment is considerably

cheaper than switching between threads belonging to different processes. The overheads

associated with thread switching are related to scheduling (choosing the next thread to

run) and context switching.

A processor context comprises the values of the processor registers such as the

program counter, and the current hardware protection domain: the address space and the

processor protection mode (supervisor or user). A context switch is the transition

between contexts that takes place when switching between threads, or when a single

thread makes a system call or takes another type of exception. It involves the following:

• the saving of the processor’s original register state, and the loading of the new

state;

• in some cases, a transfer to a new protection domain – this is known as a domain

transition.

Switching between threads sharing the same execution environment entirely at user

level involves no domain transition and is relatively cheap. Switching to the kernel, or

to another thread belonging to the same execution environment via the kernel, involves

a domain transition. The cost is therefore greater but it is still relatively low if the kernel

is mapped into the process’s address space. When switching between threads belonging

to different execution environments, however, there are greater overheads. The box

below explains the expensive implications of hardware caching for these domain

The aliasing problem • Memory management units usually include a hardware

cache to speed up the translation between virtual and physical addresses, called a

translation lookaside buffer (TLB). TLBs, and also virtually addressed data and

instruction caches, suffer in general from the so-called aliasing problem. The same

virtual address can be valid in two different address spaces, but in general it is

supposed to refer to different physical data in the two spaces. Unless their entries are

tagged with a context identifier, TLBs and virtually addressed caches are unaware of

this and so might contain incorrect data. Therefore the TLB and cache contents have

to be flushed on a switch to a different address space. Physically addressed caches do

not suffer from the aliasing problem but using virtual addresses for cache lookups is

a common practice, largely because it allows the lookups to be overlapped with

address translation.SECTION 7.4 PROCESSES AND THREADS 313

transitions. Longer-term costs of having to acquire hardware cache entries and main

memory pages are more liable to apply when such a domain transition occurs. Figures

quoted by Anderson et al. [1991] are 1.8 milliseconds for the Topaz kernel to switch

between UNIX processes and 0.4 milliseconds to switch between threads belonging to

the same execution environment. Even lower costs (0.04 milliseconds) are achieved if

threads are switched at user level. These figures are given as a rough guide only; they

do not measure the longer-term caching costs.

In the example above of the client process with two threads, the first thread

generates data and passes it to the second thread, which makes a remote method

invocation or remote procedure call. Since the threads share an address space, there is

no need to use message passing to pass the data. Both threads may access the data via a

common variable. Herein lies both the advantage and the danger of using multi-threaded

processes. The convenience and efficiency of access to shared data is an advantage. This

is particularly so for servers, as the example of caching file data given above showed.

However, threads that share an address space and that are not written in a type-safe

language are not protected from one another. An errant thread can arbitrarily alter data

used by another thread, causing a fault. If protection is required, then either a type-safe

language should be used or it may be preferable to use multiple processes instead of

multiple threads.

Threads programming • Threads programming is concurrent programming, as

traditionally studied in, for example, the field of operating systems. This section refers

to the following concurrent programming concepts, which are explained fully by Bacon

[2002]: race conditions, critical sections (Bacon calls these critical regions), monitors,

condition variables and semaphores.

Much threads programming is done in a conventional language, such as C, that has

been augmented with a threads library. The C Threads package developed for the Mach

operating system is an example of this. More recently, the POSIX Threads standard

IEEE 1003.1c-1995, known as pthreads, has been widely adopted. Boykin et al. [1993]

describe both C Threads and pthreads in the context of Mach.

Some languages provide direct support for threads, including Ada95 [Burns and

Wellings 1998], Modula-3 [Harbison 1992] and Java [Oaks and Wong 1999]. We give

an overview of Java threads here.

Like any threads implementation, Java provides methods for creating threads,

destroying them and synchronizing them. The Java Thread class includes the

constructor and management methods listed in Figure 7.8. The Thread and Object

synchronization methods are in Figure 7.9.

Thread lifetimes • A new thread is created on the same Java virtual machine (JVM) as

its creator, in the SUSPENDED state. After it is made RUNNABLE with the start()

method, it executes the run() method of an object designated in its constructor. The JVM

and the threads on top of it all execute in a process on top of the underlying operating

system. Threads can be assigned a priority, so that a Java implementation that supports

priorities will run a particular thread in preference to any thread with lower priority. A

thread ends its life when it returns from the run() method or when its destroy() method

is called.

Programs can manage threads in groups. Every thread belongs to one group,

which it is assigned at the time of its creation. Thread groups are useful when several314 CHAPTER 7 OPERATING SYSTEM SUPPORT

applications coexist on the same JVM. One example of their use is security: by default,

a thread in one group cannot perform management operations on a thread in another

group. For example, an application thread cannot mischievously interrupt a system

windowing (AWT) thread.

Thread groups also facilitate control of the relative priorities of threads (on Java

implementations that support priorities). This is useful for browsers running applets and

for web servers running programs called servlets [Hunter and Crawford 1998], which

create dynamic web pages. An unprivileged thread within an applet or servlet can only

create a new thread that belongs to its own group, or to a descendant group created

within it (the exact restrictions depend upon the SecurityManager in place). Browsers

and servers can assign threads belonging to different applets or servlets to different

groups and set the maximum priority of each group as a whole (including descendant

groups). There is no way for an applet or servlet thread to override the group priorities

set by the manager threads, since they cannot be overridden by calls to setPriority().

Thread synchronization • Programming a multi-threaded process requires great care.

The main difficult issues are the sharing of objects and the techniques used for thread

coordination and cooperation. Each thread’s local variables in methods are private to it

– threads have private stacks. However, threads are not given private copies of static

(class) variables or object instance variables.

Consider, for example, the shared queues that we described earlier in this section,

which I/O threads and worker threads use to transfer requests in some server threading

architectures. Race conditions can in principle arise when threads manipulate data

Figure 7.8 Java thread constructor and management methods

Thread(ThreadGroup group, Runnable target, String name)

Creates a new thread in the SUSPENDED state, which will belong to group and be

identified as name; the thread will execute the run() method of target.

setPriority(int newPriority), getPriority()

Sets and returns the thread’s priority.

run()

A thread executes the run() method of its target object, if it has one, and otherwise its

own run() method (Thread implements Runnable).

start()

Changes the state of the thread from SUSPENDED to RUNNABLE.

sleep(long millisecs)

Causes the thread to enter the SUSPENDED state for the specified time.

yield()

Causes the thread to enter the READY state and invokes the scheduler.

destroy()

Destroys the thread.SECTION 7.4 PROCESSES AND THREADS 315

structures such as queues concurrently. The queued requests can be lost or duplicated

unless the threads’ pointer manipulations are carefully coordinated.

Java provides the synchronized keyword for programmers to designate the wellknown monitor construct for thread coordination. Programmers designate either entire

methods or arbitrary blocks of code as belonging to a monitor associated with an

individual object. The monitor’s guarantee is that at most one thread can execute within

it at any time. We could serialize the actions of the I/O and worker threads in our

example by designating addTo() and removeFrom() methods in the Queue class as

synchronized methods. All accesses to variables within those methods would then be

carried out in mutual exclusion with respect to invocations of these methods.

Java allows threads to be blocked and woken up via arbitrary objects that act as

condition variables. A thread that needs to block awaiting a certain condition calls an

object’s wait() method. All objects implement this method, since it belongs to Java’s

root Object class. Another thread calls notify() to unblock at most one thread or

notifyAll() to unblock all threads waiting on that object. Both notification methods also

belong to the Object class.

As an example, when a worker thread discovers that there are no requests to

process, it calls wait() on the instance of Queue. When the I/O thread subsequently adds

a request to the queue, it calls the queue’s notify() method to wake up a worker.

The Java synchronization methods are given in Figure 7.9. In addition to the

synchronization primitives that we have mentioned, the join() method blocks the caller

until the target thread’s termination. The interrupt() method is useful for prematurely

waking a waiting thread. All the standard synchronization primitives, such as

semaphores, can be implemented in Java. But care is required, since Java’s monitor

guarantees apply only to an object’s synchronized code; a class may have a mixture of

synchronized and non-synchronized methods. Note also that the monitor implemented

by a Java object has only one implicit condition variable, whereas in general a monitor

may have several condition variables.

Thread scheduling • An important distinction is between preemptive and nonpreemptive scheduling of threads. In preemptive scheduling, a thread may be suspended

at any point to make way for another thread, even when the preempted thread would

Figure 7.9 Java thread synchronization calls

thread.join(long millisecs)

Blocks the calling thread for up to the specified time or until thread has terminated.

thread.interrupt()

Interrupts thread: causes it to return from a blocking method call such as sleep().

object.wait(long millisecs, int nanosecs)

Blocks the calling thread until a call made to notify() or notifyAll() on object wakes

the thread, the thread is interrupted or the specified time has elapsed.

object.notify(), object.notifyAll()

Wakes, respectively, one or all of any threads that have called wait() on object.316 CHAPTER 7 OPERATING SYSTEM SUPPORT

otherwise continue running. In non-preemptive scheduling (sometimes called coroutine

scheduling), a thread runs until it makes a call to the threading system (for example, a

system call), when the system may deschedule it and schedule another thread to run.

The advantage of non-preemptive scheduling is that any section of code that does

not contain a call to the threading system is automatically a critical section. Race

conditions are thus conveniently avoided. On the other hand, non-preemptively

scheduled threads cannot take advantage of a multiprocessor, since they run exclusively.

Care must be taken over long-running sections of code that do not contain calls to the

threading system. The programmer may need to insert special yield() calls, whose sole

function is to enable other threads to be scheduled and make progress. Nonpreemptively scheduled threads are also unsuited to real-time applications, in which

events are associated with absolute times by which they must be processed.

Java does not, by default, support real-time processing, although real-time

implementations exist [www.rtj.org]. For example, multimedia applications that process

data such as voice and video have real-time requirements for both communication and

processing (e.g., filtering and compression) [Govindan and Anderson 1991]. Chapter 20

will examine real-time thread-scheduling requirements. Process control is another

example of a real-time domain. In general, each real-time domain has its own threadscheduling requirements. It is therefore sometimes desirable for applications to

implement their own scheduling policies. To consider this, we turn now to the

implementation of threads.

Threads implementation • Many kernels provide native support for multi-threaded

processes, including Windows, Linux, Solaris, Mach and Mac OS X. These kernels

provide thread-creation and -management system calls, and they schedule individual

threads. Some other kernels have only a single-threaded process abstraction. Multithreaded processes must then be implemented in a library of procedures linked to

application programs. In such cases, the kernel has no knowledge of these user-level

threads and therefore cannot schedule them independently. A threads runtime library

organizes the scheduling of threads. A thread would block the process, and therefore all

threads within it, if it made a blocking system call, so the asynchronous (non-blocking)

I/O facilities of the underlying kernel are exploited. Similarly, the implementation can

utilize the kernel-provided timers and software interrupt facilities to timeslice between

threads.

When no kernel support for multi-threaded processes is provided, a user-level

threads implementation suffers from the following problems:

• The threads within a process cannot take advantage of a multiprocessor.

• A thread that takes a page fault blocks the entire process and all threads within it.

• Threads within different processes cannot be scheduled according to a single

scheme of relative prioritization.

User-level threads implementations, on the other hand, have significant advantages over

kernel-level implementations:

• Certain thread operations are significantly less costly. For example, switching

between threads belonging to the same process does not necessarily involve a

system call, which entails a relatively expensive trap to the kernel.SECTION 7.4 PROCESSES AND THREADS 317

• Given that the thread-scheduling module is implemented outside the kernel, it can

be customized or changed to suit particular application requirements. Variations

in scheduling requirements occur largely because of application-specific

considerations such as the real-time nature of multimedia processing.

• Many more user-level threads can be supported than could reasonably be provided

by default by a kernel.

It is possible to combine the advantages of user-level and kernel-level threads

implementations. One approach, applied, for example, to the Mach kernel [Black 1990],

is to enable user-level code to provide scheduling hints to the kernel’s thread scheduler.

Another, adopted in the Solaris 2 operating system, is a form of hierarchical scheduling.

Each process creates one or more kernel-level threads, known in Solaris as ‘lightweight

processes’. User-level threads are also supported. A user-level scheduler assigns each

user-level thread to a kernel-level thread. This scheme can take advantage of

multiprocessors, and also benefits because some thread-creation and thread-switching

operations take place at user level. The scheme’s disadvantage is that it still lacks

flexibility: if a thread blocks in the kernel, then all user-level threads assigned to it are

also prevented from running, regardless of whether they are eligible to run.

Several research projects have developed hierarchical scheduling further in order

to provide greater efficiency and flexibility. These include work on so-called scheduler

activations [Anderson et al. 1991], the multimedia work of Govindan and Anderson

[1991], the Psyche multiprocessor operating system [Marsh et al. 1991], the Nemesis

kernel [Leslie et al. 1996] and the SPIN kernel [Bershad et al. 1995]. The insight driving

these designs is that what a user-level scheduler requires from the kernel is not just a set

of kernel-supported threads onto which it can map user-level threads. The user-level

scheduler also requires the kernel to notify it of the events that are relevant to its

scheduling decisions. We describe the scheduler activations design in order to make this

clear.

The FastThreads package of Anderson et al. [1991] is an implementation of a

hierarchic, event-based scheduling system. They consider the main system components

to be a kernel running on a computer with one or more processors, and a set of

application programs running on it. Each application process contains a user-level

scheduler, which manages the threads inside the process. The kernel is responsible for

allocating virtual processors to processes. The number of virtual processors assigned to

a process depends on such factors as the applications’ requirements, their relative

priorities and the total demand on the processors. Figure 7.10(a) shows an example of a

three-processor machine, on which the kernel allocates one virtual processor to process

A, running a relatively low-priority job, and two virtual processors to process B. They

are virtual processors because the kernel can allocate different physical processors to

each process as time goes by, while keeping its guarantee of how many processors it has

allocated.

The number of virtual processors assigned to a process can also vary. Processes

can give back a virtual processor that they no longer need; they can also request extra

virtual processors. For example, if process A has requested an extra virtual processor and

B terminates, then the kernel can assign one to A.318 CHAPTER 7 OPERATING SYSTEM SUPPORT

Figure 7.10(b) shows that a process notifies the kernel when either of two types of

event occurs: when a virtual processor is ‘idle’ and no longer needed, or when an extra

virtual processor is required.

Figure 7.10(b) also shows that the kernel notifies the process when any of four

types of event occurs. A scheduler activation (SA) is a call from the kernel to a process,

which notifies the process’s scheduler of an event. Entering a body of code from a lower

layer (the kernel) in this way is sometimes called an upcall. The kernel creates an SA by

loading a physical processor’s registers with a context that causes it to commence

execution of code in the process, at a procedure address designated by the user-level

scheduler. An SA is thus also a unit of allocation of a timeslice on a virtual processor.

The user-level scheduler has the task of assigning its READY threads to the set of SAs

currently executing within it. The number of those SAs is at most the number of virtual

processors that the kernel has assigned to the process.

The four types of event that the kernel notifies the user-level scheduler (which we

shall refer to simply as ‘the scheduler’) of are as follows:

Virtual processor allocated: The kernel has assigned a new virtual processor to the

process, and this is the first timeslice upon it; the scheduler can load the SA with the

context of a READY thread, which can thus recommence execution.

SA blocked: An SA has blocked in the kernel, and the kernel is using a fresh SA to

notify the scheduler; the scheduler sets the state of the corresponding thread to

BLOCKED and can allocate a READY thread to the notifying SA.

SA unblocked: An SA that was blocked in the kernel has become unblocked and is

ready to execute at user level again; the scheduler can now return the corresponding

thread to the READY list. In order to create the notifying SA, the kernel either

allocates a new virtual processor to the process or preempts another SA in the same

process. In the latter case, it also communicates the preemption event to the

scheduler, which can reevaluate its allocation of threads to SAs.

Process

A

Process

B

Virtual processors Kernel

Process

Kernel

P idle

P needed

P added

SA blocked

SA unblocked

SA preempted

Figure 7.10 Scheduler activations

A. Assignment of virtual processors

to processes

B. Events between user-level scheduler & kernel

Key: P = processor; SA = scheduler activationSECTION 7.5 COMMUNICATION AND INVOCATION 319

SA preempted: The kernel has taken away the specified SA from the process

(although it may do this to allocate a processor to a fresh SA in the same process);

the scheduler places the preempted thread in the READY list and reevaluates the

thread allocation.

This hierarchical scheduling scheme is flexible because the process’s user-level

scheduler can allocate threads to SAs in accordance with whatever policies can be built

on top of the low-level events. The kernel always behaves the same way. It has no

influence on the user-level scheduler’s behaviour, but it assists the scheduler through its

event notifications and by providing the register state of blocked and preempted threads.

The scheme is potentially efficient because no user-level thread need stay in the READY

state if there is a virtual processor on which to run it.

7.5 Communication and invocation

Here we concentrate on communication as part of the implementation of what we have

called an invocation – a construct, such as a remote method invocation, remote

procedure call or event notification, whose purpose is to bring about an operation on a

resource in a different address space.

We cover operating system design issues and concepts by asking the following

questions about the OS:

• What communication primitives does it supply?

• Which protocols does it support and how open is the communication implementation?

• What steps are taken to make communication as efficient as possible?

• What support is provided for high-latency and disconnected operation?

We focus on the first two questions here then turn to the final two in Sections 7.5.1 and

7.5.2, respectively.

Communication primitives • Some kernels designed for distributed systems have

provided communication primitives tailored to the types of invocation that Chapter 5

described. Amoeba [Tanenbaum et al. 1990], for example, provides doOperation,

getRequest and sendReply as primitives. Amoeba, the V system and Chorus provide

group communication primitives. Placing relatively high-level communication

functionality in the kernel has the advantage of efficiency. If, for example, middleware

provides RMI over UNIX’s connected (TCP) sockets, then a client must make two

communication system calls (socket write and read) for each remote invocation. Over

Amoeba, it would require only a single call to doOperation. The savings in system call

overhead are liable to be even greater with group communication.

In practice, middleware, and not the kernel, provides most high-level

communication facilities found in systems today, including RPC/RMI, event

notification and group communication. Developing such complex software as user-level

code is much simpler than developing it for the kernel. Developers typically implement

middleware over sockets giving access to Internet standard protocols – often connected320 CHAPTER 7 OPERATING SYSTEM SUPPORT

sockets using TCP but sometimes unconnected UDP sockets. The principal reasons for

using sockets are portability and interoperability: middleware is required to operate over

as many widely used operating systems as possible, and all common operating systems,

such as UNIX and the Windows family, provide similar socket APIs giving access to

TCP and UDP protocols.

Despite the widespread use of TCP and UDP sockets provided by common

kernels, research continues to be carried out into lower-cost communication primitives

in experimental kernels. We examine performance issues further in Section 7.5.1.

Protocols and openness • One of the main requirements of the operating system is to

provide standard protocols that enable interworking between middleware

implementations on different platforms. Several research kernels developed in the 1980s

incorporated their own network protocols tuned to RPC interactions – notably Amoeba

RPC [van Renesse et al. 1989], VMTP [Cheriton 1986] and Sprite RPC [Ousterhout et

al. 1988]. However, these protocols were not widely used beyond their native research

environments. By contrast, the designers of the Mach 3.0 and Chorus kernels (as well as

L4 [Härtig et al. 1997]) decided to leave the choice of networking protocols entirely

open. These kernels provide message passing between local processes only, and leave

network protocol processing to a server that runs on top of the kernel.

Given the everyday requirement for access to the Internet, compatibility at the

level of TCP and UDP is required of operating systems for all but the smallest of

networked devices. And the operating system is still required to enable middleware to

take advantage of novel low-level protocols. For example, users want to benefit from

wireless technologies such as infrared and radio frequency (RF) transmission,

preferably without having to upgrade their applications. This requires that

corresponding protocols, such as IrDA for infrared networking and Bluetooth or IEEE

802.11 for RF networking, can be integrated.

Protocols are normally arranged in a stack of layers (see Chapter 3). Many

operating systems allow new layers to be integrated statically, by including a layer such

as IrDA as a permanently installed protocol ‘driver’. By contrast, dynamic protocol

composition is a technique whereby a protocol stack can be composed on the fly to meet

the requirements of a particular application, and to utilize whichever physical layers are

available given the platform’s current connectivity. For example, a web browser running

on a notebook computer should be able to take advantage of a wide area wireless link

while the user is on the road, and then a faster Ethernet or IEEE 802.11 connection when

the user is back in the office.

Another example of dynamic protocol composition is use of a customized requestreply protocol over a wireless networking layer, to reduce round-trip latencies. Standard

TCP implementations have been found to work poorly over wireless networking media

[Balakrishnan et al. 1996], which tend to exhibit higher rates of packet loss than wired

media. In principle, a request-response protocol such as HTTP could be engineered to

work more efficiently between wirelessly connected nodes by using the wireless

transport layer directly, rather than using an intermediate TCP layer.

Support for protocol composition appeared in the design of the UNIX Streams

facility [Ritchie 1984], in Horus [van Renesse et al. 1995] and in the x-kernel

[Hutchinson and Peterson 1991]. A more recent example is the construction of a

configurable transport protocol CTP on top of the Cactus system for dynamic protocol

composition [Bridges et al. 2007].SECTION 7.5 COMMUNICATION AND INVOCATION 321

7.5.1 Invocation performance

Invocation performance is a critical factor in distributed system design. The more

designers separate functionality between address spaces, the more remote invocations

are required. Clients and servers may make many millions of invocation-related

operations in their lifetimes, so small fractions of milliseconds count in invocation costs.

Network technologies continue to improve, but invocation times have not decreased in

proportion with increases in network bandwidth. This section will explain how software

overheads often predominate over network overheads in invocation times – at least, for

the case of a LAN or intranet. This is in contrast to a remote invocation over the Internet

– for example, fetching a web resource. On the Internet, network latencies are highly

variable and relatively high on average; throughput may be relatively low, and server

load often predominates over per-request processing costs. For an example of latencies,

Bridges et al. [2007] report minimal UDP message round-trips taking average times of

about 400 milliseconds over the Internet between two computers connected across US

geographical regions, as opposed to about 0.1 milliseconds when identical computers

were connected over a single Ethernet.

RPC and RMI implementations have been the subject of study because of the

widespread acceptance of these mechanisms for general-purpose client-server

processing. Much of the research has been carried out into invocations over the network,

and particularly into how invocation mechanisms can take advantage of highperformance networks [Hutchinson et al. 1989, van Renesse et al. 1989, Schroeder and

Burrows 1990, Johnson and Zwaenepoel 1993, von Eicken et al. 1995, Gokhale and

Schmidt 1996]. There is also, as we shall show, an important special case of RPCs

between processes hosted at the same computer [Bershad et al. 1990, 1991].

Invocation costs • Calling a conventional procedure or invoking a conventional

method, making a system call, sending a message, remote procedure calling and remote

method invocation are all examples of invocation mechanisms. Each mechanism causes

code to be executed outside the scope of the calling procedure or object. Each involves,

in general, the communication of arguments to this code and the return of data values to

the caller. Invocation mechanisms can be either synchronous, as for example in the case

of conventional and remote procedure calls, or asynchronous.

The important performance-related distinctions between invocation mechanisms,

apart from whether or not they are synchronous, are whether they involve a domain

transition (that is, whether they cross an address space), whether they involve

communication across a network and whether they involve thread scheduling and

switching. Figure 7.11 shows the particular cases of a system call, a remote invocation

between processes hosted at the same computer, and a remote invocation between

processes at different nodes in the distributed system.

Invocation over the network • A null RPC (and similarly, a null RMI) is defined as an

RPC without parameters that executes a null procedure and returns no values. Its

execution involves an exchange of messages carrying some system data but no user data.

The time taken by a null RPC between user processes connected by a LAN is on the

order of a tenth of a millisecond (see, for example, measurements by Bridges et al.

[2007] of round-trip UDP times using two 2.2GHz Pentium 3 Xeon PCs across a 100

megabits/second Ethernet). By comparison, a null conventional procedure call takes a322 CHAPTER 7 OPERATING SYSTEM SUPPORT

small fraction of a microsecond. Approximately 100 bytes in total are passed across the

network for a null RPC. With a raw bandwidth of 100 megabits/second, the total

network transfer time for this amount of data is about 0.01 milliseconds. Clearly, much

of the observed delay – the total RPC call time experienced by a client – has to be

accounted for by the actions of the operating system kernel and user-level RPC runtime

code.

Null invocation (RPC, RMI) costs are important because they measure a fixed

overhead, the latency. Invocation costs increase with the sizes of arguments and results,

but in many cases the latency is significant compared with the remainder of the delay.

Consider an RPC that fetches a specified amount of data from a server. It has one

integer request argument, specifying how much data to return. It has two reply

arguments, an integer specifying success or failure (the client might have given an

invalid size) and, when the call is successful, an array of bytes from the server.

Figure 7.11 Invocations between address spaces

Control transfer via

trap instruction

User Kernel

Thread

User 1 User 2

Control transfer via

privileged instructions

Thread 1 Thread 2

Protection domain

boundary

(a) System call

(b) RPC/RMI (within one computer)

(c) RPC/RMI (between computers)

Kernel

User 1 User 2

Thread 1 Network Thread 2

Kernel 1 Kernel 2SECTION 7.5 COMMUNICATION AND INVOCATION 323

Figure 7.12 shows, schematically, client delay against requested data size. The

delay is roughly proportional to the size until the size reaches a threshold at about

network packet size. Beyond that threshold, at least one extra packet has to be sent, to

carry the extra data. Depending on the protocol, a further packet might be used to

acknowledge this extra packet. Jumps in the graph occur each time the number of

packets increases.

Delay is not the only figure of interest for an RPC implementation: RPC

throughput (or bandwidth) is also of concern when data has to be transferred in bulk.

This is the rate of data transfer between computers in a single RPC. If we examine Figure

7.12, we can see that the throughput is relatively low for small amounts of data, when

the fixed processing overheads predominate. As the amount of data is increased, the

throughput rises as those overheads become less significant.

Recall that the steps in an RPC are as follows (RMI involves similar steps):

• A client stub marshals the call arguments into a message, sends the request

message and receives and unmarshals the reply.

• At the server, a worker thread receives the incoming request, or an I/O thread

receives the request and passes it to a worker thread; in either case, the worker

calls the appropriate server stub.

• The server stub unmarshals the request message, calls the designated procedure,

and marshals and sends the reply.

The following are the main components accounting for remote invocation delay, besides

network transmission times:

Marshalling: Marshalling and unmarshalling, which involve copying and converting

data, create a significant overhead as the amount of data grows.

Figure 7.12 RPC delay against parameter size

1000 2000

RPC delay

Requested data

size (bytes)

Packet

size

0324 CHAPTER 7 OPERATING SYSTEM SUPPORT

Data copying: Potentially, even after marshalling, message data is copied several

times in the course of an RPC:

1. across the user–kernel boundary, between the client or server address space and

kernel buffers;

2. across each protocol layer (for example, RPC/UDP/IP/Ethernet);

3. between the network interface and kernel buffers.

Transfers between the network interface and main memory are usually handled by

direct memory access (DMA). The processor handles the other copies.

Packet initialization: This involves initializing protocol headers and trailers,

including checksums. The cost is therefore proportional, in part, to the amount of data

sent.

Thread scheduling and context switching: These may occur as follows:

1. Several system calls (that is, context switches) are made during an RPC, as

stubs invoke the kernel’s communication operations.

2. One or more server threads is scheduled.

3. If the operating system employs a separate network manager process, then each

Send involves a context switch to one of its threads.

Waiting for acknowledgements: The choice of RPC protocol may influence delay,

particularly when large amounts of data are sent.

Careful design of the operating system can help reduce some of these costs. The case

study of the Firefly RPC design available at www.cdk5.net/oss shows some of these in

detail, as well as techniques that are applicable within the middleware implementation.

We have already shown how appropriate operating system support for threads can

help reduce multi-threading overheads. The operating system can also have an impact

in reducing memory-copying overheads through memory-sharing facilities.

Memory sharing • Shared regions (introduced in Section 7.4) may be used for rapid

communication between a user process and the kernel, or between user processes. Data

is communicated by writing to and reading from the shared region. Data is thus passed

efficiently, without being copied to and from the kernel’s address space. But system

calls and software interrupts may be required for synchronization, such as when the user

process has written data that should be transmitted, or when the kernel has written data

for the user process to consume. Of course, a shared region is only justified if it is used

sufficiently to offset the initial cost of setting it up.

Even with shared regions, the kernel still has to copy data from the buffers to the

network interface. The U-Net architecture [von Eicken et al. 1995] even allows userlevel code to have direct access to the network interface itself, so that user-level code

can transfer the data to the network without any copying.

Choice of protocol • The delay that a client experiences during request-reply interactions over TCP is not necessarily worse than for UDP and in fact is sometimes better,

particularly for large messages. However, care is required when implementing requestreply interactions on top of a protocol such as TCP, which was not specifically designedSECTION 7.5 COMMUNICATION AND INVOCATION 325

for this purpose. In particular, TCP’s buffering behaviour can hinder good performance,

and its connection overheads put it at a disadvantage compared with UDP, unless

enough requests are made over a single connection to render the overhead per request

negligible.

The connection overheads of TCP are particularly evident in web invocations.

HTTP 1.0, now relatively little-used, makes a separate TCP connection for every

invocation. Client browsers are delayed while the connection is made. Furthermore,

TCP’s slow-start algorithm has the effect of delaying the transfer of HTTP data

unnecessarily in many cases. The slow-start algorithm operates pessimistically in the

face of possible network congestion by allowing only a small window of data to be sent

at first, before an acknowledgement is received. Nielsen et al. [1997] discuss how HTTP

1.1, now widely used instead of HTTP 1.0, makes use of so-called persistent

connections, which last over the course of several invocations. The initial connection

costs are thus amortized, as long as several invocations are made to the same web server.

This is likely, as users often fetch several pages from the same site, each containing

several images.

Nielsen et al. also found that overriding the operating system’s default buffering

behaviour could have a significant impact on the invocation delay. It is often beneficial

to collect several small messages and then send them together, rather than sending them

in separate packets, because of the per-packet latency that we described above. For this

reason, the OS does not necessarily dispatch data over the network immediately after the

corresponding socket write() call. The default OS behaviour is to wait until its buffer is

full or to use a timeout as the criterion for dispatching the data over the network, in the

hope that more data will arrive.

Nielsen et al. found that in the case of HTTP 1.1 the default operating system

buffering behaviour could cause significant unnecessary delays because of the timeouts.

To remove these delays, they altered the kernel’s TCP settings and forced network

dispatch on HTTP request boundaries. This is a good example of how an operating

system can help or hinder middleware because of the policies it implements.

Invocation within a computer • Bershad et al. [1990] report a study that showed that, in

the installation examined, most cross-address-space invocation took place within a

computer and not, as might be expected in a client-server installation, between

computers. The trend towards placing service functionality inside user-level servers

means that more and more invocations will be to a local process. This is especially so as

caching is pursued aggressively if the data needed by a client is liable to be held in a local

server. The cost of an RPC within a computer is growing in importance as a system

performance parameter. These considerations suggest that this local case should be

optimized.

Figure 7.11 suggests that a cross-address-space invocation is implemented within

a computer exactly as it is between computers, except that the underlying message

passing happens to be local. Indeed, this has often been the model implemented. Bershad

et al. developed a more efficient invocation mechanism for the case of two processes on

the same machine called lightweight RPC (LRPC). The LRPC design is based on

optimizations concerning data copying and thread scheduling.

First, they noted that it would be more efficient to use shared memory regions for

client-server communication, with a different (private) region between the server and326 CHAPTER 7 OPERATING SYSTEM SUPPORT

each of its local clients. Such a region contains one or more A (for argument) stacks (see

Figure 7.13). Instead of RPC parameters being copied between the kernel and user

address spaces involved, the client and server are able to pass arguments and return

values directly via an A stack. The same stack is used by the client and server stubs. In

LRPC, arguments are copied once: when they are marshalled onto the A stack. In an

equivalent RPC, they are copied four times: from the client stub’s stack onto a message;

from the message to a kernel buffer, from the kernel buffer to a server message, and from

the message to the server stub’s stack. There may be several A stacks in a shared region,

because several threads in the same client may call the server at the same time.

Bershad et al. also considered the cost of thread scheduling. Compare the model

of system call and remote procedure calls in Figure 7.11. When a system call occurs,

most kernels do not schedule a new thread to handle the call but instead perform a

context switch on the calling thread so that it handles the system call. In an RPC, a

remote procedure may exist in a different computer from the client thread, so a different

thread must be scheduled to execute it. In the local case, however, it may be more

efficient for the client thread – which would otherwise be BLOCKED – to call the

invoked procedure in the server’s address space.

A server must be programmed differently in this case to the way we have

described servers before. Instead of setting up one or more threads, which then listen on

ports for invocation requests, the server exports a set of procedures that it is prepared to

have called. Threads in local processes may enter the server’s execution environment as

long as they start by calling one of the server’s exported procedures. A client needing to

invoke a server’s operations must first bind to the server interface (not shown in the

figure). It does this via the kernel, which notifies the server; when the server has

responded to the kernel with a list of allowed procedure addresses, the kernel replies to

the client with a capability for invoking the server’s operations.

An invocation is shown in Figure 7.13. A client thread enters the server’s

execution environment by first trapping to the kernel and presenting it with a capability.

Figure 7.13 A lightweight remote procedure call

1. Copy args

2. Trap to kernel

4. Execute procedure

and copy results

Client

User stub

Server

Kernel

stub

3. Upcall 5. Return (trap)

A

A stackSECTION 7.5 COMMUNICATION AND INVOCATION 327

The kernel checks this and only allows a context switch to a valid server procedure; if it

is valid, the kernel switches the thread’s context to call the procedure in the server’s

execution environment. When the procedure in the server returns, the thread returns to

the kernel, which switches the thread back to the client execution environment. Note that

clients and servers employ stub procedures to hide the details just described from

application writers.

Discussion of LRPC • There is little doubt that LRPC is more efficient than RPC for the

local case, as long as enough invocations take place to offset the memory management

costs. Bershad et al. [1990] record LRPC delays a factor of three smaller than those of

RPCs executed locally.

Location transparency is not sacrificed in Bershad’s implementation. A client stub

examines a bit set at bind time that records whether the server is local or remote, and

proceeds to use LRPC or RPC, respectively. The application is unaware of which is

used. However, migration transparency might be hard to achieve when a resource is

transferred from a local server to a remote server or vice versa, because of the need to

change invocation mechanisms.

In later work, Bershad et al. [1991] describe several performance improvements,

which are addressed particularly to multiprocessor operation. The improvements largely

concern avoiding traps to the kernel and scheduling processors in such a way as to avoid

unnecessary domain transitions. For example, if a processor is idling in the server’s

memory management context at the time a client thread attempts to invoke a server

procedure, then the thread should be transferred to that processor. This avoids a domain

transition; at the same time, the client’s processor may be reused by another thread in

the client. These enhancements involve an implementation of two-level (user and

kernel) thread scheduling, as described in Section 7.4.

7.5.2 Asynchronous operation

We have discussed how the operating system can help the middleware layer to provide

efficient remote invocation mechanisms. But in the Internet environment the effects of

relatively high latencies, low throughput and high server loads may outweigh any

benefits that the OS can provide. We can add to this the phenomena of network

disconnection and reconnection, which can be regarded as causing extremely highlatency communication. Users’ mobile computers are not connected to the network all

the time. Even if they have wide area wireless access (for example, using cellular

communication), they may be peremptorily disconnected when, for example, their train

enters a tunnel.

A common technique to defeat high latencies is asynchronous operation, which

arises in two programming models: concurrent invocations and asynchronous

invocations. These models are largely in the domain of middleware rather than operating

system kernel design, but it is useful to consider them here, while we are examining the

topic of invocation performance

Making invocations concurrently • In the first model, the middleware provides only

blocking invocations, but the application spawns multiple threads to perform blocking

invocations concurrently.328 CHAPTER 7 OPERATING SYSTEM SUPPORT

A good example of such an application is a web browser. A web page typically

contains several images and may contain many. The browser does not need to obtain the

images in a particular sequence, so it makes several concurrent requests at a time. That

way, the time taken to complete all the image requests is typically lower than the delay

that would result from making the requests serially. Not only is the total communication

delay less, in general, but the browser can overlap computation such as image rendering

with communication.

Figure 7.14 shows the potential benefits of interleaving invocations (such as

HTTP requests) between a client and a single server on a single-processor machine. In

the serialized case, the client marshals the arguments, calls the Send operation and then

waits until the reply from the server arrives – whereupon it Receives, unmarshals and

then processes the results. After this it can make the second invocation.

In the concurrent case, the first client thread marshals the arguments and calls the

Send operation. The second thread then immediately makes the second invocation. Each

thread waits to receive its results. The total time taken is liable to be lower than in the

serialized case, as the figure shows. Similar benefits apply if the client threads make

Figure 7.14 Times for serialized and concurrent invocations

Client Server

execute request

Send

Receive

unmarshal

marshal

Receive

unmarshal

process results

marshal

Send

process args

marshal

Send

process args

transmission

Receive

unmarshal

process results

execute request

Send

Receive

unmarshal

marshal

marshal

Send

process args

marshal

Send

process args

execute request

Send

Receive

unmarshal

marshal

execute request

Send

Receive

unmarshal

marshal

Receive

unmarshal

process results

Receive

unmarshal

process results

time

Client Server

Serialized invocations Concurrent invocationsSECTION 7.5 COMMUNICATION AND INVOCATION 329

concurrent requests to several servers, and if the client executes on a multiprocessor

even greater throughput is potentially possible, since the two threads’ processing can

also be overlapped.

Returning to the particular case of HTTP, the study by Nielsen et al. [1997] that

we referred to above also measured the effects of concurrently interleaved HTTP 1.1

invocations (which they call pipelining) over persistent connections. They found that

pipelining reduced network traffic and could lead to performance benefits for clients, as

long as the operating system provides a suitable interface for flushing buffers, to

override the default TCP behaviour.

Asynchronous invocations • An asynchronous invocation is one that is performed

asynchronously with respect to the caller. That is, it is made with a non-blocking call,

which returns as soon as the invocation request message has been created and is ready

for dispatch.

Sometimes the client does not require any response (except perhaps an indication

of failure if the target host could not be reached). For example, CORBA oneway

invocations have maybe semantics. Otherwise, the client uses a separate call to collect

the results of the invocation. For example, the Mercury communication system [Liskov

and Shrira 1988] supports asynchronous invocations. An asynchronous operation

returns an object called a promise. Eventually, when the invocation succeeds or is

deemed to have failed, the Mercury system places the status and any return values in the

promise. The caller uses the claim operation to obtain the results from the promise. The

claim operation blocks until the promise is ready, whereupon it returns the results or

exceptions from the call. The ready operation is available for testing a promise without

blocking – it returns true or false according to whether the promise is ready or blocked.

Persistent asynchronous invocations • Traditional asynchronous invocation mechanisms such as Mercury invocations and CORBA oneway invocations are implemented

upon TCP streams and fail if a stream breaks – that is, if the network link is down or the

target host crashes.

But a more developed form of the asynchronous invocation model, which we shall

call persistent asynchronous invocation, is becoming increasingly relevant because of

disconnected operation. This model is similar to Mercury in terms of the programming

operations it provides, but the difference is in its failure semantics. A conventional

invocation mechanism (synchronous or asynchronous) is designed to fail after a given

number of timeouts have occurred, but these short-term timeouts are often not

appropriate where disconnections or very high latencies occur.

A system for persistent asynchronous invocation tries indefinitely to perform the

invocation, until it is known to have succeeded or failed, or until the application cancels

the invocation. An example is Queued RPC (QRPC) in the Rover toolkit for mobile

information access [Joseph et al. 1997].

As its name suggests, QRPC queues outgoing invocation requests in a stable log

while there is no network connection and schedules their dispatch over the network to

servers when there is a connection. Similarly, it queues invocation results from servers

in what we can consider to be the client’s invocation ‘mailbox’ until the client reconnects and collects them. Requests and results may be compressed when they are

queued, before their transmission over a low-bandwidth network.330 CHAPTER 7 OPERATING SYSTEM SUPPORT

QRPC can take advantage of different communication links for sending an

invocation request and receiving the reply. For example, a request could be dispatched

over a cellular data link while the user is on the road, and then the response delivered

over an Ethernet link when the user connects her device to the corporate intranet. In

principle, the invocation system can even store the invocation results near to the user’s

next expected point of connection.

The client’s network scheduler operates according to various criteria and does not

necessarily dispatch invocations in FIFO order. Applications can assign priorities to

individual invocations. When a connection becomes available, QRPC evaluates its

bandwidth and the expense of using it. It dispatches high-priority invocation requests

first, and may not dispatch all of them if the link is slow and expensive (such as a wide

area wireless connection), assuming that a faster, cheaper link such as an Ethernet will

become available eventually. Similarly, QRPC takes priority into account when fetching

invocation results from the mailbox over a low-bandwidth link.

Programming with an asynchronous invocation system (persistent or otherwise)

raises the issue of how users can continue using the applications on their client device

while the results of invocations are still not known. For example, the user may wonder

whether they have succeeded in updating a paragraph in a shared document, or if

someone else has made a conflicting update, such as deleting the paragraph. Chapter 18

examines this issue.

7.6 Operating system architecture

In this section, we examine the architecture of a kernel suitable for a distributed system.

We adopt a first-principles approach of starting with the requirement of openness and

examining the major kernel architectures that have been proposed, with this in mind.

An open distributed system should make it possible to:

• run only that system software at each computer that is necessary for it to carry out

its particular role in the system architecture – system software requirements can

vary between, for example, mobile phones and server computers, and loading

redundant modules wastes memory resources;

• allow the software (and the computer) implementing any particular service to be

changed independently of other facilities;

• allow for alternatives of the same service to be provided, when this is required to

suit different users or applications;

• introduce new services without harming the integrity of existing ones.

The separation of fixed resource management mechanisms from resource management

policies, which vary from application to application and service to service, has been a

guiding principle in operating system design for a long time [Wulf et al. 1974]. For

example, we said that an ideal scheduling system would provide mechanisms that enable

a multimedia application such as video conferencing to meet its real-time demands

while coexisting with a non-real-time application such as web browsing.SECTION 7.6 OPERATING SYSTEM ARCHITECTURE 331

Ideally, the kernel would provide only the most basic mechanisms upon which the

general resource management tasks at a node are carried out. Server modules would be

dynamically loaded as required, to implement the required resource management

policies for the currently running applications.

Monolithic kernels and microkernels • There are two key examples of kernel design: the

so-called monolithic and microkernel approaches. These designs differ primarily in the

decision as to what functionality belongs in the kernel and what is to be left to server

processes that can be dynamically loaded to run on top of it. Although microkernels have

not been deployed widely, it is instructive to understand their advantages and

disadvantages compared with the typical kernels found today.

The UNIX operating system kernel has been called monolithic (see definition in

the box below). This term is meant to suggest that it is massive – it performs all basic

operating system functions and takes up in the order of megabytes of code and data –

and that it is undifferentiated, i.e. it is coded in a non-modular way. The result is that to

a large extent it is intractable: altering any individual software component to adapt it to

changing requirements is difficult. Another example of a monolithic kernel is that of the

Sprite network operating system [Ousterhout et al. 1988]. A monolithic kernel can

contain some server processes that execute within its address space, including file

servers and some networking. The code that these processes execute is part of the

standard kernel configuration (see Figure 7.15).

By contrast, in the case of a microkernel design the kernel provides only the most

basic abstractions, principally address spaces, threads and local interprocess

communication; all other system services are provided by servers that are dynamically

loaded at precisely those computers in the distributed system that require them (Figure

7.15). Clients access these system services using the kernel’s message-based invocation

mechanisms.

Figure 7.15 Monolithic kernel and microkernel

Monolithic kernel Microkernel

Server: Dynamically Kernel code and data: loaded server program:

.......

.......

Key:

.......

S4

S1 S2 S3

S1 S2 S3 S4

Monolithic • The Chambers 20th Century Dictionary gives the following definition

of monolith and monolithic. monolith, n. a pillar, or column, of a single stone:

anything resembling a monolith in uniformity, massiveness or intractability. – adj.

monolithic pertaining to or resembling a monolith: of a state, an organization, etc.,

massive, and undifferentiated throughout: intractable for this reason.332 CHAPTER 7 OPERATING SYSTEM SUPPORT

We said above that users are liable to reject operating systems that do not run their

applications. But in addition to extensibility, microkernel designers have another goal:

the binary emulation of standard operating systems such as UNIX [Armand et al. 1989,

Golub et al. 1990, Härtig et al. 1997].

The place of the microkernel – in its most general form – in the overall distributed

system design is shown in Figure 7.16. The microkernel appears as a layer between the

hardware layer and a layer consisting of major system components called subsystems. If

performance is the main goal, rather than portability, then middleware may use the

facilities of the microkernel directly. Otherwise, it uses a language runtime support

subsystem, or a higher-level operating system interface provided by an operating system

emulation subsystem. Each of these, in turn, is implemented by a combination of library

procedures linked into applications and a set of servers running on top of the

microkernel.

There can be more than one system call interface – more than one ‘operating

system’ – presented to the programmer on the same underlying platform. An example is

the implementation of UNIX and OS/2 on top of the Mach distributed operating system

kernel. Note that operating system emulation is different from machine virtualization

(see Section 7.7).

Comparison • The chief advantages of a microkernel-based operating system are its

extensibility and its ability to enforce modularity behind memory protection boundaries.

In addition, a relatively small kernel is more likely to be free of bugs than one that is

larger and more complex.

The advantage of a monolithic design is the relative efficiency with which

operations can be invoked. System calls may be more expensive than conventional

procedures, but even using the techniques we examined in the previous section, an

invocation to a separate user-level address space on the same node is more costly still.

The lack of structure in monolithic designs can be avoided by the use of software

engineering techniques such as layering, used in MULTICS [Organick 1972], or objectoriented design, used for example in Choices [Campbell et al. 1993]. Windows employs

a combination of both [Custer 1998]. But Windows remains ‘massive’, and the majority

of its functionality is not designed to be routinely replaceable. Even a modularized large

Figure 7.16 The role of the microkernel

Middleware

Language

support

subsystem A

Language

support

subsystem B

OS emulation

subsystem

....

Microkernel

Hardware

The microkernel supports middleware via subsystemsSECTION 7.6 OPERATING SYSTEM ARCHITECTURE 333

kernel can be hard to maintain, and it provides limited support for an open distributed

system. As long as modules are executed within the same address space, using a

language such as C or C++ that compiles to efficient code but permits arbitrary data

accesses, it is possible for strict modularity to be broken by programmers seeking

efficient implementations, and for a bug in one module to corrupt the data in another.

Some hybrid approaches • Two of the original microkernels, Mach [Acetta et al. 1986]

and Chorus [Rozier et al. 1990], began their developmental life running servers only as

user processes. In this configuration, modularity is hardware-enforced through address

spaces. Where servers require direct access to hardware, special system calls can be

provided for these privileged processes, which map device registers and buffers into

their address spaces. The kernel turns interrupts into messages, which enables user-level

servers to handle interrupts.

Because of performance problems, the Chorus and Mach microkernel designs

eventually changed to allow servers to be loaded dynamically either into the kernel

address space or into a user-level address space. In each case, clients interact with

servers using the same interprocess communication calls. A developer can thus debug a

server at user level and then, when the development is deemed complete, allow the

server to run inside the kernel’s address space in order to optimize system performance.

But such a server then threatens the integrity of the system, should it turn out still to

contain bugs.

The SPIN operating system design [Bershad et al. 1995] finesses the problem of

trading off efficiency for protection by employing language facilities for protection. The

kernel and all dynamically loaded modules grafted onto the kernel execute within a

single address space. But all are written in a type-safe language (Modula-3), so they can

be mutually protected. Protection domains within the kernel address space are

established using protected name spaces. No module grafted onto the kernel may access

a resource unless it has been handed a reference for it, and Modula-3 enforces the rule

that a reference can only be used to perform operations allowed by the programmer.

In an attempt to minimize the dependencies between system modules, the SPIN

designers chose an event-based model as a mechanism for interaction between modules

grafted into the kernel’s address space (see Section 6.3 for a discussion of event-based

programming). The system defines a set of core events, such as network packet arrival,

timer interrupts, page fault occurrences and thread state changes. System components

operate by registering themselves as handlers for the events that affect them. For

example, a scheduler would register itself to handle events similar to those we studied

in the scheduler activations system in Section 7.4.

Operating systems such as Nemesis [Leslie et al. 1996] exploit the fact that, even

at the hardware level, an address space is not necessarily also a single protection domain.

The kernel coexists in a single address space with all dynamically loaded system

modules and all applications. When it loads an application, the kernel places the

application’s code and data in regions chosen from those that are available at runtime.

The advent of processors with 64-bit addressing has made single-address-space

operating systems particularly attractive, since they support very large address spaces

that can accommodate many applications.

The kernel of a single-address-space operating system sets the protection

attributes on individual regions within the address space to restrict access by user-level334 CHAPTER 7 OPERATING SYSTEM SUPPORT

code. User-level code still runs with the processor in a particular protection context

(determined by settings in the processor and memory management unit), which gives it

full access to its own regions and only selectively shared access to others. The saving of

a single address space, compared with using multiple address spaces, is that the kernel

need never flush any caches when it implements a domain transition.

Some later kernel designs, such as L4 [Härtig et al. 1997] and the Exokernel

[Kaashoek et al. 1997], take the approach that what we have described as ‘microkernels’

still contain too much policy as opposed to mechanism. L4 is a ‘second-generation’

microkernel design that forces dynamically loaded system modules to execute in userlevel address spaces, but optimizes interprocess communication to offset the costs of

doing so. It offloads much of the kernel’s complexity by delegating the management of

address spaces to user-level servers. The Exokernel takes a quite different approach,

employing user-level libraries instead of user-level servers to supply functional

extensions. It provides protected allocation of extremely low-level resources such as

disk blocks, and it expects all other resource management functionality – even a file

system – to be linked into applications as libraries.

In the words of one microkernel designer [Liedtke 1996], ‘the microkernel story

is full of good ideas and blind alleys’. As we shall see in the next section, the need to

support multiple subsystems and also enforce protection between these subsystems is

now met by the concept of virtualization which has replaced microkernel approaches as

the key innovation in operating system design.

7.7 Virtualization at the operating system level

Virtualization is an important concept in distributed systems. We have already seen one

application of virtualization in the context of networking, in the form of overlay

networks (see Section 4.5) offering support for particular classes of distributed

application. Virtualization is also applied in the context of operating systems; indeed, it

is in this context that virtualization has had the most impact. In this section, we examine

what it means to apply virtualization at the operating system level (system

virtualization) and also present a case study of Xen, a leading example of system-level

virtualization.

7.7.1 System virtualization

The goal of system virtualization is to provide multiple virtual machines (virtual

hardware images) over the underlying physical machine architecture, with each virtual

machine running a separate operating system instance. The concept stems from the

observation that modern computer architectures have the necessary performance to

support potentially large numbers of virtual machines and multiplex resources between

them. Multiple instances of the same operating system can run on the virtual machines

or a range of different operating systems can be supported. The virtualization system

allocates the physical processor(s) and other resources of a physical machine between

all virtual machines that it supports.SECTION 7.7 VIRTUALIZATION AT THE OPERATING SYSTEM LEVEL 335

Historically, processes were used to share the processor and other resources

between multiple tasks running on behalf of one or several users. System virtualization

has emerged more recently and is now commonly used for this purpose. It offers benefits

for security and clean separation of tasks and in allocating and charging each user for

their use of resources more precisely than can be achieved with processes running in a

single system.

To fully understand the motivation for virtualization at the operating system level,

it is useful to consider different use cases of the technology:

• On server machines, an organization assigns each service it offers to a virtual

machine and then optimally allocates the virtual machines to physical servers.

Unlike processes, virtual machines can be migrated quite simply to other physical

machines, adding flexibility in managing the server infrastructure. This approach

has the potential to reduce investment in server computers and to reduce energy

consumption, a key issue for large server farms.

• Virtualization is very relevant to the provision of cloud computing. As described

in Chapter 1, cloud computing adopts a model where storage, computation and

higher-level objects built over them are offered as a service. The services offered

range from low-level aspects such as physical infrastructure (referred to as

infrastructure as a service), through software platforms such as the Google App

Engine, featured in Chapter 21, (platform as a service), to arbitrary applicationlevel services (software as a service). Indeed, the first is enabled directly by

virtualization, allowing users of the cloud to be provided with one or more virtual

machines for their own use.

• The developers of virtualization solutions are also motivated by the need for

distributed applications to create and destroy virtual machines readily and with

little overhead. This is required in applications that may need to demand resources

dynamically, such as multiplayer online games or distributed multimedia

applications, as featured in Chapter 1 [Whitaker et al. 2002]. Support for such

applications can be enhanced by adopting appropriate resource allocation policies

to meet quality of service requirements of virtual machines.

• A quite different case arises in providing convenient access to several different

operating system environments on a single desktop computer. Virtualization can

be used to provide multiple operating system types on one physical architecture.

For example, on a Macintosh OS X computer, the Parallels Desktop virtual

machine monitor enables a Windows or a Linux system to be installed and to

coexist with OS X, sharing the underlying physical resources.

System virtualization is implemented by a thin layer of software on top of the underlying

physical machine architecture; this layer is referred to as a virtual machine monitor or

hypervisor. This virtual machine monitor provides an interface based closely on the

underlying physical architecture. More precisely, in full virtualization the virtual

machine monitor offers an identical interface to the underlying physical architecture.

This has the advantage that existing operating systems can run transparently and

unmodified on the virtual machine monitor. Experience has shown, however, that full

virtualization can be hard to realize with satisfactory performance on many computer

architectures, including the x86 family of processors, and that performance may be336 CHAPTER 7 OPERATING SYSTEM SUPPORT

improved by allowing a modified interface to be provided (with the drawback that

operating systems then need to be ported to this modified interface). This technique is

known as paravirtualization and is considered in more detail in the case study below.

Note that virtualization is quite distinct from the microkernel approach as

discussed in Section 7.6. Although microkernels support the co-existence of multiple

operating systems, this is achieved by emulating the operating system on top of the reusable building blocks offered by the microkernel. In contrast, in operating system

virtualization, an operating system is run directly (or with minor modifications) on the

virtualized hardware. The key advantage of virtualization and the principal reason for its

predominance over microkernels is that applications can run in virtualized environments

without being rewritten or recompiled.

Virtualization began with the IBM 370 architecture, whose VM operating system

can present several complete virtual machines to different programs running at the same

computer. The technique can therefore be traced back to the 1970s. More recently, there

has been an explosion in interest in virtualization, with a number of research projects and

commercial systems providing virtualization solutions for commodity PCs, servers and

cloud infrastructure. Examples of leading virtualization solutions include Xen [Barham

et al. 2003a], Denali [Whitaker et al. 2002], VMWare, Parallels and Microsoft Virtual

Server. We present a case study of the Xen approach below.

7.7.2 Case study: The Xen approach to system virtualization

Xen is a leading example of system virtualization, initially developed as part of the

Xenoserver project at the Computer Laboratory, Cambridge University and now

maintained by an open source community [www.xen.org]. An associated company,

XenSource, was acquired by Citrix Systems in 2007 and Citrix now offers enterprise

solutions based on Xen technology, including XenoServer and associated management

and automation tools. The description of Xen provided below is based on the paper by

Barham et al. [2003a] and associated XenoServer internal reports [Barham et al. 2003b,

Fraser et al. 2003], together with an excellent and comprehensive book about the

internals of the Xen hypervisor [Chisnall 2007].

The overall goal of the XenoServer project [Fraser et al. 2003] is to provide a

public infrastructure for wide area distributed computing. As such, this is an early

example of cloud computing focussing on infrastructure as a service. In the XenoServer

vision, the world is populated by XenoServers that are capable of executing code on

behalf of customers who are then billed for the resources they use.

The two main outputs of the project are the Xen virtual machine monitor and the

XenoServer Open Platform discussed in more detail below.

The Xen virtual machine monitor • Xen is a virtual machine monitor that was designed

initially to support the implementation of XenoServers but has since evolved into a

standalone solution to system virtualization. The goal of Xen is to enable multiple

operating system instances to run in complete isolation on conventional hardware with

minimal performance overhead introduced by the associated virtualization. Xen is

designed to scale to very large numbers of operating system instances (up to several

hundred virtual machines on a single machine) and deal with heterogeneity, seeking to

support most major operating systems, including Windows, Linux, Solaris and NetBSD.EXERCISES 337

Xen also runs on a variety of hardware platforms including 32- and 64-bit x86

architectures and also PowerPC and IA64 CPUs.

The architecture of Xen: The overall architecture of Xen is captured in Figure 7.17. The

Xen virtual machine monitor (known as the hypervisor) is central to this architecture,

supporting the virtualization of the underlying physical resources, specifically the CPU

and its instruction set, the scheduling of the CPU resource and the physical memory. The

overall goal of the hypervisor is to provide virtual machines with a virtualization of the

hardware, providing the appearance that each virtual machine has its own (virtualized)

physical machine and multiplexing the virtual resources onto the underlying physical

resources. In achieving this, it must also ensure strong protection between the different

virtual machines it supports.

The hypervisor follows the design of Exokernel (introduced in Section 7.6) by

implementing a minimal set of mechanisms for resource management and isolation and

leaving higher level policy to other parts of the systems architecture – in particular the

domains, as discussed below. The hypervisor also has no knowledge of devices or their

management but rather just provides a conduit for interacting with devices (as again

discussed below). This minimal design is important for two key reasons:

• Xen is primarily concerned with isolation, including isolation of faults, and yet a

fault in the hypervisor can crash the whole system. It is therefore important that

the hypervisor is minimal, thoroughly tested and bug-free.

• The hypervisor represents an inevitable overhead relative to executing on the bare

hardware, and it is important for the performance of the system that this is as

lightweight as possible (as we shall see below, paravirtualization also helps to

minimize this overhead through bypassing the hypervisor wherever possible).

Figure 7.17 The architecture of Xen

Underlying physical hardware (for example x86 machine)

Domain

control

interface

0

Virtual

x86 CPU Scheduling . . .

Guest OS

(XenoLinux

Device drivers

Control

plane

software

Domain0

Guest OS

(XenoBSD)

Device drivers

User

software

DomainU

Guest OS

(XenoLinux)

Device drivers

software

. . .

User

Xen

hypervisor

)

Virtual

memory338 CHAPTER 7 OPERATING SYSTEM SUPPORT

The role of the hypervisor is to support a potentially large number of virtual machine

instances (called domains in Xen) all running guest operating systems. The guest

operating systems run in a set of domains referred to collectively as domainU, or the

unprivileged domain, referring to their lack of privilege in terms of accessing physical

(as opposed to virtual) resources. In other words, all access to resources is carefully

controlled by Xen. Xen also supports a special domain, referred to as domain0, that has

privileged access to hardware resources and acts as a control plane for the Xen

architecture providing a clean separation between mechanism and policy in the system

(we will see examples of the usage of domain0 below). Domain0 is configured to run a

Xen port of Linux (XenoLinux), whereas other domains can run any guest operating

system. Note that the Xen architecture allows selected privileges to be granted to

domainU, specifically the ability to access hardware devices directly or to create new

domains. In practice, though, the most common configuration is for domain0 to retain

these privileges.

To continue our study of Xen, we consider the implementation of the core

functions of the hypervisor – namely, the virtualization of the underlying hardware

(including the use of paravirtualization), scheduling and virtual memory management –

before showing how Xen supports the management of devices, We conclude by

considering what it takes to port a given operating system to Xen.

Virtualization of the underlying CPU: The primary role of the hypervisor is to provide each

domain with a virtualization of the underlying CPU, that is provide the appearance that

each domain has its own (virtual) CPU and associated instruction set. The complexity

of this step depends entirely on the architecture of the given CPU. In this section, we

focus particularly on virtualization as it applies to the x86 architecture, the dominant

processor family in use today.

Popek and Goldberg [1974], in a classic paper on virtualization requirements,

focus on all instructions that can change the state of the machine in a way that can impact

on other processes (sensitive instructions), further subdividing such instructions into:

• control-sensitive instructions that attempt to change the configuration of resources

in the system, for example changing virtual memory mappings;

• behaviour-sensitive instructions that read privileged state and through this reveal

physical rather than virtual resources, thus breaking the virtualization.

They then state that a condition for being virtualizable is that all sensitive instructions

(control- and behaviour-sensitive) must be intercepted by the hypervisor (or equivalent

kernel mechanism). More specifically, this is achieved by trapping into the hypervisor,

supported by the concept of privileged instructions in a machine architecture – that is,

instructions that either execute in privileged mode or generate a trap (which can then

take them into privileged mode). This leads to the following precise statement of the

Popek and Goldberg condition:

Condition for virtualization : A processor architecture lends itself to virtualization if

all sensitive instructions are privileged instructions.

Unfortunately, in the x86 family of processors, this is not the case: it is possible to

identify 17 instructions that are sensitive but not privileged. For example, the LAR (load

access rights) and LSL (load segment limit) instructions fall into this category. TheySECTION 7.7 VIRTUALIZATION AT THE OPERATING SYSTEM LEVEL 339

need to be trapped by the hypervisor to ensure correct virtualization, but there is no

mechanism to do this as they are not privileged.

One solution is to provide a layer of emulation for all instructions in the instruction

set. It is then possible to manage sensitive instructions within this layer. This is what is

done in full virtualization and this approach has the advantage that guest operating

systems can run unchanged in this virtualized environment. However, this approach can

be expensive, adding cost to every affected instruction call. Paravirtualization, in

contrast, takes the view that many instructions can run directly on the bare hardware

without emulation and that privileged instructions should be trapped and dealt with by

the hypervisor. This then leaves the sensitive instructions that are not privileged; a

paravirtualization solution recognizes that such instructions can lead to potential

problems but leaves this to be dealt with in the guest operating system. In other words,

the guest operating system must be rewritten to tolerate or deal with any side effects of

these instructions. One approach, for example, is to rewrite sections of the code to avoid

the usage of problematic instructions. This paravirtualization approach greatly improves

the performance of virtualization, but at the expense of requiring a port of the guest

operating system to the virtualized environment.

To understand the implementation of paravirtualization further, it is helpful to

look at levels (or rings) of privilege in modern processors. For example, the x86 family

supports four levels of privilege with ring 0 being the most privileged, ring 1 being the

next most privileged and so on, with ring 3 being the least privileged. In a traditional

operating system environment, the kernel will run in ring 0 and applications in ring 3

with rings 1 and 2 unused. Traps take the flow of control from the application to the

kernel and allow privileged activities to take place. In Xen, the hypervisor runs in ring

0 and this is the only ring that can execute privileged instructions. The guest operating

systems then run in ring 1, with applications running in ring 3. Privileged instructions

are rewritten as hypercalls that trap into the hypervisor allowing the hypervisor to

control execution of these potentially sensitive operations. All other sensitive

instructions must be managed by the guest operating system, as discussed above.

This distinction between kernel-based operating systems and Xen is summarized

in Figure 7.18.

Hypercalls are asynchronous and hence represent notifications that the

corresponding instructions should be executed (there is no blocking in the guest

operating system awaiting a result). Communication between the hypervisor and the

guest operating system is also asynchronous and is supported by a simple event

mechanism offered by the Xen hypervisor. This is used, for example, to deal with device

interrupts. The hypervisor maps such hardware interrupts to software events targeted

towards the right guest operating system. The Xen hypervisor is therefore completely

event-driven.

Scheduling: We saw in Section 7.4 that many operating system environments support

two levels of scheduling – that is, the scheduling of processes and the subsequent

scheduling of user-level threads within processes. Xen goes one step further,

introducing an extra level of scheduling concerned with the execution of particular guest

operating systems. It achieves this by introducing the concept of a virtual CPU (VCPU),

with each VCPU supporting a guest system. Scheduling therefore involves the following

steps:340 CHAPTER 7 OPERATING SYSTEM SUPPORT

• The hypervisor schedules VCPUs on to the underlying physical CPU(s), thereby

providing each guest with a portion of the underlying physical processing time.

• Guest operating systems schedule kernel-level threads on to their allocated

VCPU(s).

• Where applicable, threads libraries in user space schedule user-level threads onto

the available kernel-level threads.

The key requirement in Xen is that the design of the underlying hypervisor scheduler is

predictable, as higher-level schedulers will make assumptions about the behaviour of

this scheduler and it is crucial that these assumptions are met.

Xen supports two schedulers, Simple EDF and the Credit Scheduler:

Xen’s Simple Earliest Deadline First (SEDF) Scheduler: This scheduler operates

by selecting the VCPU that has the closest deadline, with deadlines calculated

according to two parameters, n (the slice) and m (the period). For example, a domain

can request 10 ms (n) every 100 ms (m). The deadline is defined as the latest time this

domain can be run to meet its deadline. Returning to our example, at the start point

of the system, this VCPU can be scheduled as late as 90 ms into the 100 ms period

and still meet its deadline. The scheduler operates by picking the earliest of the

current deadlines, looking across the set of runnable VCPUs.

Xen’s Credit Scheduler: For this scheduler, each domain (VCPU) is specified in

terms of two properties, the weight and the cap. The weight determines the share of

the CPU that should be given to that VCPU. For example, if one VCPU has a weight

of 64 and another a weight of 32, the first VCPU should get double the share of the

second. Legal weights range from 1 to 65535, with the default being 256. This

Figure 7.18 Use of rings of privilege

ring 0

ring 3

ring 2

ring 1

ring 0

ring 3

ring 2

ring 1

kernel

applications applications

hypervisor guest OS

a) kernel-based operating systems b) paravirtualization in XenSECTION 7.7 VIRTUALIZATION AT THE OPERATING SYSTEM LEVEL 341

behaviour is modified by the cap, which expresses the total percentage of the CPU

that should be given to the corresponding VCPU, expressed as a percentage. This can

be left as uncapped. The scheduler transforms the weight associated with a VCPU

into credits, and as the VCPU runs, it consumes credits. The VCPU is deemed under

if it has credits left; otherwise it is deemed over. For each CPU, the scheduler

maintains a queue of runnable VCPUs, with all the under VCPUs first, followed by

the over VCPUs. When a VCPU is unscheduled, it is placed in this queue at the end

of the appropriate category (depending upon whether it is now under or over credit).

The scheduler then picks the first element in the queue to run next. As a form of load

balancing, if a given CPU has no under VCPUs, it will search the queues of other

CPUs for a possible candidate to schedule.

These replace previous Xen schedulers, including a simple round robin scheduler, one

based on borrowed virtual time (designed to provide a proportional share of the

underlying CPU based on setting different domain weights), and Atropos (designed to

support soft real-time scheduling). Further details of these schedulers can be found in

Chisnall [2007].

It is also possible to add new schedulers to the Xen hypervisor, but this is

something that should be done with caution and after extensive testing because of the

requirements of the hypervisor as discussed above. Chisnall [2007] provides a step-bystep guide on how to implement such a simple scheduler in Xen.

Interaction between guest operating systems and the underlying scheduler is via a

number of scheduler-specific hypercalls, including operations to voluntary yield the

CPU (but remain runnable), to block a particular domain until an event has occurred or

to shutdown the domain for a specified reason.

Virtual memory management: Virtual memory management is the most complicated

aspect of virtualization partly because of the complexity of underlying hardware

solutions to memory management and partly because of the need to inject extra levels

of protection to provide isolation between different domains. We provide below some

general principles of memory management in Xen. The reader is encouraged to study

the detailed description of virtual memory management in Xen provided by Chisnall

[2007].

The overall approach to virtualization of memory management in Xen is captured

in Figure 7.19. As with scheduling, Xen adopts a three-level architecture with the

hypervisor managing physical memory, the kernel of the guest operating system

providing pseudo-physical memory and applications within that operating system

provided with virtual memory, as would be expected of any underlying operating

system. The concept of pseudo-physical memory is crucial to understanding this

architecture and is described further below.

The key design decision in the virtual memory management architecture is to keep

the functionality of the hypervisor to a minimum. The hypervisor effectively has just

two roles – the allocation and subsequent management of physical memory in the form

of pages:

• In terms of memory allocation, the hypervisor retains a small portion of physical

memory for its own needs and then allocates pages to domains on demand. For

example, when a new domain is created, it is given a set of pages according to its

declared needs. In practice, this set of pages will be fragmented across the physical342 CHAPTER 7 OPERATING SYSTEM SUPPORT

address space, and this may be in conflict with the expectations of the guest

operating system (which may expect a contiguous address space). The role of the

pseudo-physical memory is to provide this abstraction by offering a contiguous

pseudo-physical address space and then maintaining a mapping from this address

space to the real physical addresses. Crucially, this mapping must be managed by

the guest operating system and not in the hypervisor, to maintain the lightweight

nature of the hypervisor (more specifically, the composition of the two functions

shown in Figure 7.19 is carried out in the guest). This approach allows the guest

operating system to interpret the mapping in its own context (for example, for

some guest operating systems where contiguous addresses are not expected, this

mapping can be eliminated) and also makes it easier to migrate a domain to a

different address space, for example in server consolidation. The same mechanism

is also used to support suspension and resumption of guest operating systems. On

suspension, the state of the domain is serialized to disk, and on resumption, this

state is restored but in a different physical location. This is supported by the extra

level of indirection in the memory management architecture.

• In terms of managing the physical memory, the hypervisor exports a small set of

hypercalls to manipulate the underlying page tables. As an illustration, the

hypercall pt\_update(list of requests) is used by a guest operating system to request

a batch of incremental updates to a page table. This allows the hypervisor to

validate all the requests and carry out only those updates that are deemed safe, for

example to enforce isolation.

The overall result is a flexible approach to virtual memory management that allows

guest operating systems to optimize their implementation for different processor

families.

Figure 7.19 Virtualization of memory management

Virtual

Pseudo-physical

Physical

Application

Guest OS

HypervisorSECTION 7.7 VIRTUALIZATION AT THE OPERATING SYSTEM LEVEL 343

Device management: The Xen approach to device management relies on the concept of

split device drivers, as shown in Figure 7.20. As can be seen from this figure, access to

the physical device is controlled exclusively by domain0, which also hosts a real device

driver for this device. As domain0 runs XenoLinux, this will be an available Linux

device driver. It is important to stress that while some device drivers have good support

for multiplexing, others do not, and hence it is important for Xen to provide an

abstraction whereby each guest operating system can have the appearance of its own

virtual device. This is achieved by the split driver structure involving a back-end device

driver running in domain0 and a front-end driver running in the guest operating system.

The two then communicate to provide the necessary device access for the guest

operating system. The respective roles of the back-end and front-end parts of the driver

are as follows:

• The back end has two critical roles to play in the architecture. Firstly, it must

manage multiplexing (in particular access from multiple guest operating systems),

especially where support is not provided in the underlying Linux driver. Secondly,

it provides a generic interface that both captures the essential functions of the

device and is neutral to the different guest operating systems that will use it. This

is made easier because operating systems already provide a number of

abstractions that effectively provide the necessary multiplexing in a neutral way,

for example reading and writing blocks to persistent storage. Higher-level

interfaces (for example, sockets) would be inappropriate as they would be too

biased towards given operating system abstractions.

Figure 7.20 Split device drivers

Hardware

Physical

device

Hypervisor

Domain0 Domain0:

Split device

driver

Real device

driver

Split device

driver

Shared

memory

Guest O344 CHAPTER 7 OPERATING SYSTEM SUPPORT

• The front end, in contrast, is very simple and acts as a proxy for the device in the

guest operating system environment, accepting commands and then

communicating with the back end as described below.

Communication between the front end and back end of the split device structure is

facilitated by the creation of a memory page shared between the two components. The

region of shared memory is established using a grant table mechanism supported by the

hypervisor. A grant table is an array of structures (grant entries) that supports operations

to provide permissions to grant foreign access to a memory reservation or to access other

memory reservations via grant references. Access can be granted to read or write the

shared memory region. This mechanism provides a lightweight and high-performance

means for different domains to communicate in Xen.

The normal mechanism to communicate is to use a data structure known as an I/O

ring in this shared memory region, which supports two-way asynchronous

communication between the two parts of the split device driver. The structure of an I/O

ring is shown in Figure 7.21. Domains communicate through requests and responses. In

particular, a domain writes its request clockwise, starting at the request start indicator

(assuming there is enough room) and moving on the pointer accordingly. The other end

can then read the data from its end, again moving the associated pointer. The same

procedure then occurs for responses. For devices that continually transfer large amounts

of data, the corresponding endpoints will poll this data structure. For less frequent

transfers, I/O rings can be supplemented by the use of the Xen event mechanism to

notify the recipient that data is ready for consumption. The mechanism for device

discovery is via a shared information space called XenStore, accessible by all domains.

XenStore is itself implemented as a device using the split device architecture, which

device drivers use to advertise their services. The information provided includes the

grant reference for the I/O rings associated with the device and also (where appropriate)

Figure 7.21 I/O rings

Request start

Request end

Response start

Response endSECTION 7.7 VIRTUALIZATION AT THE OPERATING SYSTEM LEVEL 345

any event channels associated with the device. The range of communication facilities

used by device drivers (I/O rings, events and XenStore) are collectively referred to as

XenBus.

A given Xen installation can provide different configurations of device drivers. It

is expected, though, that most Xen implementations will provide two generic drivers:

• The first one is the block device driver, offering a common abstraction onto block

devices (most commonly storage devices). The interface to this is very simple,

supporting three operations: to read or write a block and also to implement a write

barrier ensuring that all outstanding writes are completed.

• The second one is the Xen Virtual Interface Network Driver, which offers a

common interface to interact with network devices. This uses two I/O rings for

transmitting and receiving data to/from the network. Strictly speaking, the rings

are used for control flow and separate shared memory areas are used for the

associated data (which helps in terms of minimizing copies and reusing memory

regions).

Note that most of this architecture is implemented above the hypervisor, in domain0 and

the other guest operating systems. The role of the hypervisor is simply to facilitate interdomain communication, for example through the grant table mechanism, and the rest is

built on top of this minimal base. This helps considerably in terms of keeping the

hypervisor small and efficient.

Porting a guest operating system: From the descriptions above, it is now possible to see

what is required in terms of porting an operating system to the Xen environment. This

involves several key stages:

• replacing all privileged instructions used by the operating system with the relevant

hypercalls;

• taking all other sensitive instructions and reimplementing them in a way that

preserves the desired semantics of the associated operations;

• porting the virtual memory subsystem;

• developing split-level device drivers for the required set of devices, reusing the

device driver functionality already provided in domain0 together with the generic

device driver interfaces where appropriate.

This list covers the major tasks, but there are also some other, more specific tasks that

need to be carried out. For example, Xen offers its own time architecture, recognizing

the difference between real time and time passing as viewed by individual guest

operating systems.

In more detail, the hypervisor provides support for various time abstractions –

specifically, an underlying cycle counter time based on the clock of the underlying

processor and used to extrapolate other time references; domain virtual time, which

progresses at the same rate as cycle counter time but only when a particular domain is

scheduled; system time, which accurately reflects the passing of real time in the system;

and wall clock time, providing the actual time of day. It is assumed that operating system

instances running in domains will provide real time and virtual time abstractions on top

of these values, requiring further porting effort. Interestingly, both system time and wall346 CHAPTER 7 OPERATING SYSTEM SUPPORT

clock time are corrected automatically for clock drift, exploiting a single instance of an

NTP client (described in Chapter 14) running in domain0. This is just one example of

the optimizations that are enabled by the shared domain0.

The XenoServer Open Platform • As mentioned above, Xen was initially developed as

part of the XenoServer project, which investigated software infrastructure for wide area

distributed computing. We now describe the overall architecture of the associated

XenoServer Open Platform [Hand et al. 2003]. In this architecture, which is shown in

Figure 7.22, clients register with an entity known as XenoCorp in order to use the

system. The developers of the XenoServer Open Architecture anticipate a number of

competing instances of XenoCorps existing in a given system offering different

payment regimes and levels of quality of service (for example, varying support for

privacy). More formally, the role of a given XenoCorp is to offer authentication,

auditing, charging and payment services and to maintain a contractual relationship with

both clients and organizations offering XenoServers. This is supported by a process of

registration whereby identity is established, and the creation of purchase orders that

represent the commitment by the (authenticated) client to fund a given session.

In the overall architecture, XenoServers compete against each other to offer

services. The role of the XenoServer Information Service is then to allow XenoServers

to advertise their services and for clients to locate appropriate XenoServers based on

Figure 7.22 The XenoServer Open Platform Architecture

XenoCorp

XenoServer Client

Xenoserver

Information

Service

RD System

register\_client

create\_purchase\_order

register\_xenoserver

validate\_purchase\_order

charge\_from\_purchase\_order

query\_xenoserver\_status

create\_session

deploy\_task

advertise\_xenoserver lookup\_xenoserver

find\_xenoservers

updateSECTION 7.8 SUMMARY 347

their specified requirements. Advertisements are specified using XML and include

clauses covering functionality, resource availability and pricing.

The Information Service is relatively primitive, offering basic search mechanisms

over the set of advertisements. To complement this, the platform architecture also offers

a resource discovery (RD) system supporting more complex queries, such as:

• Find a XenoServer with a low-latency link to the client that meets certain resource

requirements for a given price, o.r

• Find a cluster of XenoServers that are inter-connected by low-latency links,

support secure communication and meet certain resource requirements.

The main innovation in the XenoServer project is in how it couples the above

architecture with virtualization – each XenoServer runs the Xen virtual machine

monitor, allowing clients to bid for virtual rather than physical resources and allowing

the system to manage the set of resources more effectively because of this virtualization.

This is a direct illustration of the complementary nature of cloud computing and

virtualization, as discussed above.

7.8 Summary

This chapter has described how the operating system supports the middleware layer in

providing invocations upon shared resources. The operating system provides a

collection of mechanisms upon which varying resource management policies can be

implemented, to meet local requirements and to take advantage of technological

improvements. It allows servers to encapsulate and protect resources, while allowing

clients to share them concurrently. It also provides the mechanisms necessary for clients

to invoke operations upon resources.

A process consists of an execution environment and threads: an execution

environment consists of an address space, communication interfaces and other local

resources such as semaphores; a thread is an activity abstraction that executes within an

execution environment. Address spaces need to be large and sparse in order to support

sharing and mapped access to objects such as files. New address spaces may be created

with their regions inherited from parent processes. An important technique for copying

regions is copy-on-write.

Processes can have multiple threads, which share the execution environment.

Multi-threaded processes allow us to achieve relatively cheap concurrency and to take

advantage of multiprocessors for parallelism. They are useful for both clients and

servers. Recent threads implementations allow for two-tier scheduling: the kernel

provides access to multiple processors, while user-level code handles the details of

scheduling policy.

The operating system provides basic message-passing primitives and mechanisms

for communication via shared memory. Most kernels include network communication

as a basic facility; others provide only local communication and leave network

communication to servers, which may implement a range of communication protocols.

This is a trade-off of performance against flexibility.348 CHAPTER 7 OPERATING SYSTEM SUPPORT

We discussed remote invocations and accounted for the difference between

overheads due directly to network hardware and overheads that are due to the execution

of operating system code. We found the proportion of the total time due to software to

be relatively large for a null invocation but to decrease as a proportion of the total as the

size of the invocation arguments grows. The chief overheads involved in an invocation

that are candidates for optimization are marshalling, data copying, packet initialization,

thread scheduling and context switching, as well as the flow control protocol used.

Invocation between address spaces within a computer is an important special case, and

we described the thread-management and parameter-passing techniques used in

lightweight RPC.

There are two main approaches to kernel architecture: monolithic kernels and

microkernels. The main difference between them lies in where the line is drawn between

resource management by the kernel and resource management performed by

dynamically loaded (and usually user-level) servers. A microkernel must support at least

a notion of process and interprocess communication. It supports operating system

emulation subsystems as well as language support and other subsystems, such as those

for real-time processing. Virtualization offers an attractive alternative to this style by

providing emulation of the hardware and then allowing multiple virtual machines (and

hence multiple operating systems) to coexist on the same machine.

EXERCISES

7.1 Discuss each of the tasks of the core operating system’s components – process

management, thread management, communication management, memory management

and supervisor – for UNIX (or any other OS you are familiar with). page 300

7.2 How does the combination of middleware and network operating systems enable users

to maintain a degree of autonomy for their machines? page 297

7.3 Smith decides that every thread in his processes ought to have its own protected stack –

all other regions in a process would be fully shared. Does this make sense? page 302

7.4 Do threats to a system’s integrity come only from maliciously contrived code? Discuss.

page 300

7.5 Explain how a kernel provides protection to different processes in a system. page 301

7.6 Suggest a scheme for balancing the load on a set of computers. You should discuss:

i) what user or system requirements are met by such a scheme;

ii) to what categories of applications it is suited;

iii) how to measure load and with what accuracy;

iv) how to monitor load and choose the location for a new process. Assume that

processes may not be migrated.

How would your design be affected if processes could be migrated between computers?

Would you expect process migration to have a significant cost? page 305EXERCISES 349

7.7 What are the advantages of using multi-threaded processes? Give an example of a

situation where multi-threading reduces the tendency for servers to become bottlenecks.

page 302

7.8 A file server uses caching and achieves a hit rate of 80%. File operations in the server

cost 5 ms of CPU time when the server finds the requested block in the cache, and an

additional 15 ms of disk I/O time otherwise. Explaining any assumptions you make,

estimate the server’s throughput capacity (average requests/sec) if it is:

i) single-threaded;

ii) two-threaded, running on a single processor;

iii) two-threaded, running on a two-processor computer. page 308

7.9 Threads share an execution environment. What are the different memory regions that

different processes share? page304

7.10 What are the different policies for process allocation decisions? page 305

7.11 A spin lock (see Bacon [2002]) is a boolean variable accessed via an atomic test-and-set

instruction, which is used to obtain mutual exclusion. Would you use a spin lock to

obtain mutual exclusion between threads on a single-processor computer? page 314

7.12 Explain thread-per-request and thread-per-connection architecture with respect to multithreaded servers. pages 309, 310

7.13 Does switching present a problem for threads implementation? page 312

7.14 Explain the different methods available in Java to support synchronized execution

among different threads in a system. page 315

7.15 Why should a threads package be interested in the events of a thread’s becoming

blocked or unblocked? Why should it be interested in the event of a virtual processor’s

impending preemption? (Hint: other virtual processors may continue to be allocated.)

page 318

7.16 Network transmission time accounts for 20% of a null RPC and 80% of an RPC that

transmits 1024 user bytes (less than the size of a network packet). By what percentage

will the times for these two operations improve if the network is upgraded from 10

megabits/second to 100 megabits/second? page 321

7.17 A ‘null’ RMI that takes no parameters, calls an empty procedure and returns no values

delays the caller for 2.0 milliseconds. Explain what contributes to this time.

In the same RMI system, each 1K of user data adds an extra 1.5 milliseconds. A client

wishes to fetch 32K of data from a file server. Should it use one 32K RMI or 32 1K

RMIs? page 321

7.18 Which kind of scheduling avoids race conditions? Which factors decide whether a

thread should be scheduled as non-preemptive? page 316

7.19 Explain how it is possible to combine the advantages of user-level and kernel-level

threads implementations.

page 317350 CHAPTER 7 OPERATING SYSTEM SUPPORT

7.20 i) Can a server invoked by lightweight procedure calls control the degree of

concurrency within it?

ii) Explain why and how a client is prevented from calling arbitrary code within a

server under lightweight RPC.

iii) Does LRPC expose clients and servers to greater risks of mutual interference than

conventional RPC (given the sharing of memory)? page 325

7.21 A client makes RMIs to a server. The client takes 5 ms to compute the arguments for

each request, and the server takes 10 ms to process each request. The local OS

processing time for each send or receive operation is 0.5 ms, and the network time to

transmit each request or reply message is 3 ms. Marshalling or unmarshalling takes 0.5

ms per message.

Estimate the time taken by the client to generate and return from two requests (i) if it is

single-threaded, and (ii) if it has two threads that can make requests concurrently on a

single processor. Is there a need for asynchronous RMI if processes are multi-threaded?

page 327

7.22 What is asynchronous invocation? What are the significant changes that persistent

asynchronous invocation provides over a conventional invocation mechanism? page 329

7.23 Explain the key features of a monolithic kernel. How does it differ from the microkernel

approach? page 331

7.24 How is virtualization relevant to the provision of cloud computing? page 335

7.25 On a certain computer we estimate that, regardless of the OS it runs, thread scheduling

costs about 50 μs, a null procedure call 1 ms, a context switch to the kernel 20 μs and a

domain transition 40 μs. For each of Mach and SPIN, estimate the cost to a client of

calling a dynamically loaded null procedure. page 333

7.26 What is the distinction between the virtualization approach advocated by Xen and the

style of microkernel advocated by the Exokernel project? In your answer, highlight two

things they have in common and two distinguishing characteristics between the

approaches. pages 333, 336

7.27 Sketch out in pseudo-code how you would add a simple round robin scheduler to the

Xen hypervisor using the framework discussed in Section 7.7.2. page 339

7.28 From your understanding of the Xen approach to virtualization, discuss specific features

of Xen that can support the XenoServer architecture, thus illustrating the synergy

between virtualization and cloud computing. pages 336, 346351

8

DISTRIBUTED

OBJECTS AND COMPONENTS

8.1 Introduction

8.2 Distributed objects

8.3 Case study: CORBA

8.4 From objects to components

8.5 Case studies: Enterprise JavaBeans and Fractal

8.6 Summary

A complete middleware solution must present a higher-level programming abstraction as

well as abstracting over the underlying complexities involved in distributed systems. This

chapter examines two of the most important programming abstractions, namely

distributed objects and components, and also examines associated middleware platforms

including CORBA, Enterprise JavaBeans and Fractal.

CORBA is a middleware design that allows application programs to communicate

with one another irrespective of their programming languages, their hardware and

software platforms, the networks they communicate over and their implementors.

Applications are built from CORBA objects, which implement interfaces defined in

CORBA’s interface definition language, IDL. Like Java RMI, CORBA supports transparent

invocation of methods on remote objects. The middleware component that supports RMI

is called the object request broker, or ORB.

Component-based middleware has emerged as a natural evolution of distributed

objects, providing support for managing dependencies between components, hiding lowlevel details associated with the middleware, managing the complexities of establishing

distributed applications with appropriate non-functional properties such as security, and

supporting appropriate deployment strategies. Key technologies in this area include

Enterprise JavaBeans and Fractal.352 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

8.1 Introduction

The previous chapters introduced fundamental underlying building blocks for

distributed systems in terms of communication and operating system support. This

chapter turns to complete middleware solutions, presenting distributed objects and

components as two of the most important styles of middleware in use today. This

discussion is followed in Chapters 9 and 10 with consideration of alternative approaches

based on web services and peer-to-peer solutions.

As discussed in Chapter 2, the task of middleware is to provide a higher-level

programming abstraction for the development of distributed systems and, through

layering, to abstract over heterogeneity in the underlying infrastructure to promote

interoperability and portability.

Distributed object middleware • The key characteristic of distributed objects is that they

allow you to adopt an object-oriented programming model for the development of

distributed systems and, through this, hide the underlying complexity of distributed

programming. In this approach, communicating entities are represented by objects.

Objects communicate mainly using remote method invocation, but also possibly using

an alternative communication paradigm (such as distributed events). This relatively

simple approach has a number of important benefits, including the following:

• The encapsulation inherent in object-based solutions is well suited to distributed

programming.

• The related property of data abstraction provides a clean separation between the

specification of an object and its implementation, allowing programmers to deal

solely in terms of interfaces and not be concerned with implementation details

such as the programming language and operating system used.

• This approach also lends itself to more dynamic and extensible solutions, for

example by enabling the introduction of new objects or the replacement of one

object with another (compatible) object.

A range of middleware solutions based on distributed objects are available, including

Java RMI and CORBA. We summarize the key features of distributed objects in Section

8.2 and provide a detailed case study of CORBA in Section 8.3.

Component-based middleware • Component-based solutions have been developed to

overcome a number of limitations that have been observed for application developers

working with distributed object middleware:

Implicit dependencies: Object interfaces do not describe what the implementation of

an object depends on, making object-based systems difficult to develop (especially

for third-party developers) and subsequently manage.

Programming complexity: Programming distributed object middleware leads to a

need to master many low-level details associated with middleware implementations.

Lack of separation of distribution concerns: Application developers are obliged to

consider details of concerns such as security, failure handling and concurrency,

which are largely similar from one application to another.SECTION 8.2 DISTRIBUTED OBJECTS 353

No support for deployment: Object-based middleware provides little or no support

for the deployment of (potentially complex) configurations of objects.

Component-based solutions can best be understood as a natural evolution of objectbased approaches, building on the strong heritage of this earlier work. Section 8.4

discusses this rationale in more detail and introduces the key features of a componentbased approach. Section 8.5 then presents two contrasting case studies of componentbased solutions, Enterprise JavaBeans and Fractal, with the former offering a

comprehensive solution that abstracts over many of the key issues in developing

distributed applications and the latter representing a more lightweight solution often

used to construct more complex middleware technologies.

8.2 Distributed objects

Middleware based on distributed objects is designed to provide a programming model

based on object-oriented principles and therefore to bring the benefits of the objectoriented approach to distributed programming.

Emmerich [2000] sees such distributed objects as a natural evolution from three

strands of activity:

• In distributed systems, earlier middleware was based on the client-server model

and there was a desire for more sophisticated programming abstractions.

• In programming languages, earlier work in object-oriented languages such as

Simula-67 and Smalltalk led to the emergence of more mainstream and heavily

used programming languages such as Java and C++ (languages used extensively

in distributed systems).

• In software engineering, significant progress was made in the development of

object-oriented design methods, leading to the emergence of the Unified

Modelling Language (UML) as an industrial-standard notation for specifying

(potentially distributed) object-oriented software systems.

In other words, through adopting an object-oriented approach, distributed systems

developers are not only provided with richer programming abstractions (using familiar

programming languages such as C++ and Java) but are also able to use object-oriented

design principles, tools and techniques (including UML) in the development of

distributed systems software. This represents a major step forward in an area where,

previously, such design techniques were not available. It is interesting to note that the

OMG, the organization that developed CORBA (see Section 8.3), also manages the

standardization of UML.

Distributed object middleware offers a programming abstraction based on objectoriented principles. Leading examples of distributed object middleware include Java

RMI (discussed in Section 5.5) and CORBA (examined in depth in Section 8.3 below).

While Java RMI and CORBA share a lot in common, there is one important difference:

the use of Java RMI is restricted to Java-based development, whereas CORBA is a

multi-language solution allowing objects written in a variety of languages to

interoperate. (Bindings exist for C++, Java, Python and several others.)354 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

It must be stressed that programming with distributed objects is both different

from and significantly more complex than standard object-oriented programming, as

summarized below.

The differences: Key differences between objects and distributed objects have already

been covered in Section 5.4.1, in the context of RMI. For convenience, this discussion

is repeated here in summary form (see Figure 8.1). Other differences will emerge when

we look in detail at CORBA in Section 8.3. These include:

• Class is a fundamental concept in object-oriented languages but does not feature

so prominently in distributed object middleware. As noted in the CORBA case

study, it is difficult to agree upon a common interpretation of class in a

heterogeneous environment where multiple languages coexist. In the objectoriented world more generally, class has several interpretations, including the

description of the behaviour associated with a group of objects (the template used

to create an object from the class), the place to go to instantiate an object with a

given behaviour (the associated factory) or even the group of objects that adhere

to that behaviour. While the term ‘class’ is avoided, more specific terms such as

Figure 8.1 Distributed objects

Objects Distributed objects Description of distributed object

Object references Remote object references Globally unique reference for a

distributed object; may be passed as a

parameter.

Interfaces Remote interfaces Provides an abstract specification of the

methods that can be invoked on the

remote object; specified using an

interface definition language (IDL).

Actions Distributed actions Initiated by a method invocation,

potentially resulting in invocation

chains; remote invocations use RMI.

Exceptions Distributed exceptions Additional exceptions generated from

the distributed nature of the system,

including message loss or process

failure.

Garbage collection Distributed garbage collection Extended scheme to ensure that an

object will continue to exist if at least

one object reference or remote object

reference exists for that object,

otherwise, it should be removed.

Requires a distributed garbage

collection algorithm.SECTION 8.2 DISTRIBUTED OBJECTS 355

‘factory’ and ‘template’ are readily used (a factory being an object that will

instantiate a new object from a given template).

• The style of inheritance is significantly different from that offered in most objectoriented languages. In particular, distributed object middleware offers interface

inheritance, which is a relationship between interfaces whereby the new interface

inherits the method signatures of the original interface and can add extra ones. In

contrast, object-oriented languages such as Smalltalk offer implementation

inheritance as a relationship between implementations, whereby the new class (in

this case) inherits the implementation (and hence behaviour) of the original class

and can add extra behaviour. Implementation inheritance is much more difficult

to implement, particularly in distributed systems, due to the need to resolve the

correct executable behaviour at runtime. Consider, for example, the level of

heterogeneity that may exist in a distributed system, together with the need to

implement highly scalable solutions.

Wegner, in his seminal paper on object-oriented languages [Wegner 1987], defines

object orientation as objects + class + inheritance. In distributed systems, clearly the

interpretation is somewhat different, with both class and inheritance avoided or adapted.

What remains is a strong focus on encapsulation and data abstraction, together with the

powerful links to design methodologies as emphasized above.

The added complexities: Because of the added complexities involved, the associated

distributed object middleware must provide additional functionality, as summarized

below:

Inter-object communication: A distributed object middleware framework must offer

one or more mechanisms for objects to communicate in the distributed environment.

This is normally provided by remote method invocation, although distributed object

middleware often supplements this with other communications paradigms (for

example, indirect approaches such as distributed events). CORBA provides an event

service and an associated notification service, both implemented as services on top

of the core middleware (see Section 8.3.4).

Lifecycle management: Lifecycle management is concerned with the creation,

migration and deletion of objects, with each step having to deal with the distributed

nature of the underlying environment.

Activation and deactivation: In non-distributed implementations, it can often be

assumed that objects are active all the time while the process that contains them runs.

In distributed systems, however, this cannot be assumed as the numbers of objects

may be very large, and hence it would be wasteful of resources to have all objects

available at any time. In addition, nodes hosting objects may be unavailable for

periods of time. Activation is the process of making an object active in the distributed

environment by providing the necessary resources for it to process incoming

invocations – effectively, locating the object in virtual memory and giving it the

necessary threads to execute. Deactivation is then the opposite process, rendering an

object temporarily unable to process invocations.

Persistence: Objects typically have state, and it is important to maintain this state

across possible cycles of activation and deactivation and indeed system failures.356 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

Distributed object middleware must therefore offer persistency management for

stateful objects.

Additional services: A comprehensive distributed object middleware framework

must also provide support for the range of distributed system services considered in

this book, including naming, security and transaction services.

8.3 Case study: CORBA

The Object Management Group (OMG) was formed in 1989 with a view to encouraging

the adoption of distributed object systems in order to gain the benefits of object-oriented

programming for software development and to make use of distributed systems, which

were becoming widespread. To achieve its aims, the OMG advocated the use of open

systems based on standard object-oriented interfaces. These systems would be built

from heterogeneous hardware, computer networks, operating systems and programming

languages.

An important motivation was to allow distributed objects to be implemented in

any programming language and to be able to communicate with one another. The OMG

therefore designed an interface language that was independent of any specific

implementation language.

They introduced a metaphor, the object request broker (ORB), whose role is to

help a client to invoke a method on an object (following the RMI style, as introduced in

Chapter 5). This role involves locating the object, activating the object if necessary and

then communicating the client’s request to the object, which carries it out and replies.

This section presents a case study of OMG’s Common Object Request Broker

Architecture (CORBA), building on this object request broker concept. The presentation

focuses on the CORBA 2 specification (the main innovation in its successor, CORBA

3, is the introduction of a component model, as discussed in Section 8.4).

The main components of CORBA’s language-independent RMI framework are

the following:

• an interface definition language known as IDL, which is described in some detail

in Section 8.3.1;

• an architecture, which is discussed in Section 8.3.2;

• an external data representation, called CDR, which is described in Section 4.3.1 –

it also defines specific formats for the messages in a request-reply protocol and

messages for enquiring about the location of an object, for cancelling requests and

for reporting errors;

• a standard form for remote object references, which is described in Section 8.3.3.

The CORBA architecture also allows for CORBA services – a set of generic services

that are useful for distributed applications. These are briefly introduced in Section 8.3.4

(a more complete version of this case study, including detailed consideration of CORBA

services, can be found on the companion web site [www.cdk5.net]).SECTION 8.3 CASE STUDY: CORBA 357

Section 8.3.5 also contains an example of developing a client and server using

CORBA.

For an interesting collection of articles on CORBA, see the CACM special issue

[Seetharamanan 1998].

8.3.1 CORBA RMI

Programming in a multi-language RMI system such as CORBA RMI requires more of

the programmer than programming in a single-language RMI system such as Java RMI.

The following new concepts need to be learned:

• the object model offered by CORBA;

• the interface definition language;

• its mapping onto the implementation language.

Other aspects of CORBA programming are similar to those discussed in Chapter 5. In

particular, the programmer defines remote interfaces for the remote objects and then

uses an interface compiler to produce the corresponding proxies and skeletons. But in

CORBA, proxies are generated in the client language and skeletons in the server

language.

CORBA's object model • The CORBA object model is similar to the one described in

Section 5.4.1, but clients are not necessarily objects – a client can be any program that

sends request messages to remote objects and receives replies. The term CORBA object

is used to refer to remote objects. Thus, a CORBA object implements an IDL interface,

has a remote object reference and is able to respond to invocations of methods in its IDL

interface. A CORBA object can be implemented by a language that is not objectoriented – for example, without the concept of class. Since implementation languages

will have different notions of class, or even none at all, the class concept does not exist

in CORBA (see also the discussion in Section 8.2). Therefore classes cannot be defined

in CORBA IDL, which means that instances of classes cannot be passed as arguments.

However, data structures of various types and of arbitrary complexity can be passed as

arguments.

CORBA IDL • A CORBA IDL interface specifies a name and a set of methods that

clients can request. As an initial example, Figure 8.2 shows two interfaces named Shape

(line 3) and ShapeList (line 5), which are IDL versions of the interfaces defined in Figure

5.16. These are preceded by definitions of two structs, which are used as parameter types

in defining the methods. Note in particular that GraphicalObject is defined as struct,

whereas it was a class in the Java RMI example. A component whose type is struct has

a set of fields containing values of various types, like the instance variables of an object,

but it has no methods.

In more detail, the CORBA IDL provides facilities for defining modules,

interfaces, types, attributes and method signatures. We can see examples of all of the

above, apart from modules, in Figures 5.8 and 8.2. CORBA IDL has the same lexical

rules as C++ but has additional keywords to support distribution, for example, interface,

any, attribute, in, out, inout, readonly and raises. It also allows standard C++

preprocessing facilities. See, for example, the typedef for All in Figure 8.3.358 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

The grammar of IDL is a subset of ANSI C++ with additional constructs to

support method signatures. We give here only a brief overview of IDL. A useful

overview and many examples are given in Baker [1997] and Henning and Vinoski

[1999]. The full specification is available on the OMG web site [OMG 2002a].

IDL modules: The module construct allows interfaces and other IDL type definitions to

be grouped in logical units. A module defines a naming scope, which prevents names

defined within a module from clashing with names defined outside it. For example, the

definitions of the interfaces Shape and ShapeList could belong to a module called

Whiteboard, as shown in Figure 8.3.

IDL interfaces: As we have seen, an IDL interface describes the methods that are

available in CORBA objects that implement that interface. Clients of a CORBA object

may be developed just from the knowledge of its IDL interface. From a study of our

examples, readers will see that an interface defines a set of operations and attributes and

generally depends on a set of types defined with it. For example, the PersonList interface

in Figure 5.2 defines an attribute and three methods and depends on the type Person.

IDL methods: The general form of a method signature is:

[oneway] <return\_type> <method\_name> (parameter1,..., parameterL)

[raises (except1,..., exceptN)] [context (name1,..., nameM)];

Figure 8.2 IDL interfaces Shape and ShapeList

struct Rectangle{ 1

long width;

long height;

long x;

long y;

};

struct GraphicalObject { 2

string type;

Rectangle enclosing;

boolean isFilled;

};

interface Shape { 3

long getVersion();

GraphicalObject getAllState(); // returns state of the GraphicalObject

};

typedef sequence <Shape, 100> All; 4

interface ShapeList { 5

exception FullException{ }; 6

Shape newShape(in GraphicalObject g) raises (FullException); 7

All allShapes(); // returns sequence of remote object references 8

long getVersion();

};SECTION 8.3 CASE STUDY: CORBA 359

where the expressions in square brackets are optional. For an example of a method

signature that contains only the required parts, consider:

void getPerson(in string name, out Person p);

The parameters are labelled as in, out or inout, where the value of an in parameter is

passed from the client to the invoked CORBA object and the value of an out parameter

is passed back from the invoked CORBA object to the client. Parameters labelled as

inout are seldom used, but they indicate that the parameter value may be passed in both

directions. The return type may be specified as void if no value is to be returned. Figure

5.8 illustrates a simple example of the use of those keywords. In Figure 8.2, line 7, the

parameter of newShape is an in parameter to indicate that the argument should be passed

from client to server in the request message. The return value provides an additional out

parameter – it can be indicated as void if there is no out parameter.

The parameters may be any one of the primitive types, such as long or boolean, or

one of the constructed types, such as struct or array (more information on IDL primitive

and constructed types can be found below). Our example shows the definitions of two

structs in lines 1 and 2. Sequences and arrays are defined in typedefs, as shown in line

4, which shows a sequence of elements of type Shape of length 100. The semantics of

parameter passing are as follows:

Passing CORBA objects: Any parameter whose type is specified by the name of an

IDL interface, such as the return value Shape in line 7, is a reference to a CORBA

object and the value of a remote object reference is passed.

Passing CORBA primitive and constructed types: Arguments of primitive and

constructed types are copied and passed by value. On arrival, a new value is created

in the recipient’s process. For example, the struct GraphicalObject passed as an

argument (in line 7) produces a new copy of this struct at the server.

These two forms of parameter passing are combined in the method allShapes (in line 8),

whose return type is an array of type Shape – that is, an array of remote object

references. The return value is a copy of the array in which each of the elements is a

remote object reference.

Figure 8.3 IDL module Whiteboard

module Whiteboard {

struct Rectangle{

...};

struct GraphicalObject {

...};

interface Shape {

...};

typedef sequence <Shape, 100> All;

interface ShapeList {

...};

};360 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

Invocation semantics: Remote invocation in CORBA has at-most-once call semantics as

the default. However, IDL may specify that the invocation of a particular method has

maybe semantics by using the oneway keyword. The client does not block on oneway

requests, which can be used only for methods that do not return results. For an example

of a oneway request, see the example on callbacks at the end of Section 8.3.5.

Exceptions in CORBA IDL: CORBA IDL allows exceptions to be defined in interfaces and

thrown by their methods. The optional raises expression indicates user-defined

exceptions that can be raised to terminate an execution of the method. Consider the

following example from Figure 8.2:

exception FullException{ };

Shape newShape(in GraphicalObject g) raises (FullException);

The method newShape specifies with the raises expression that it may raise an exception

called FullException, which is defined within the ShapeList interface. In our example,

the exception contains no variables. However, exceptions may be defined to contain

variables, for example:

exception FullException {GraphicalObject g};

When an exception that contains variables is raised, the server may use the variables to

return information to the client about the context of the exception.

CORBA can also produce system exceptions relating to problems with servers

(such as their being too busy or unable to activate objects), problems with

communication and client-side problems. Client programs should handle user-defined

and system exceptions. The optional context expression is used to supply mappings from

string names to string values. See Baker [1997] for an explanation of context.

IDL types: IDL supports 15 primitive types, which include short (16-bit), long (32-bit),

unsigned short, unsigned long, float (32-bit), double (64-bit), char, boolean (TRUE,

FALSE), octet (8-bit) and any (which can represent any primitive or constructed type).

Constants of most of the primitive types and constant strings may be declared using the

const keyword. IDL provides a special type called Object, whose values are remote

object references. If a parameter or result is of type Object, then the corresponding

argument may refer to any CORBA object.

IDL’s constructed types, all of which are passed by value in arguments and results,

are described in Figure 8.4. All arrays or sequences used as arguments must be defined

in typedefs. None of the primitive or constructed data types can contain references.

CORBA also supports passing non-CORBA objects by value [OMG 2002c].

These non-CORBA objects are object-like in the sense that they possess both attributes

and methods. However, they are purely local objects in that their operations cannot be

invoked remotely. The pass-by-value facility provides the ability to pass a copy of a nonCORBA object between client and server.

This is achieved by the addition to IDL of a type called valuetype for representing

non-CORBA objects. A valuetype is a struct with additional method signatures (like

those of an interface), and valuetype arguments and results are passed by value. That is,

the state is passed to the remote site and used to produce a new object at the destination.

The methods of this new object may be invoked locally, causing its state to

diverge from the state of the original object. Passing the implementation of the methodsSECTION 8.3 CASE STUDY: CORBA 361

is not so straightforward, since the client and server may use different languages.

However, if the client and server are both implemented in Java, the code can be

downloaded. For a common implementation in C++, the necessary code would need to

be present at both client and server.

This facility is useful when it is beneficial to place a copy of an object in the client

process to enable it to receive local invocations. However, it does not get us any nearer

to passing CORBA objects by value.

Attributes: IDL interfaces can have attributes as well as methods. Attributes are like

public class fields in Java. Attributes may be defined as readonly where appropriate. The

attributes are private to CORBA objects, but for each attribute declared, a pair of

accessor methods is generated automatically by the IDL compiler: one to retrieve the

value of the attribute and the other to set it. For readonly attributes, only the getter

method is provided. For example, the PersonList interface defined in Figure 5.2 includes

the following definition of an attribute:

readonly attribute string listname;

Figure 8.4 IDL constructed types

Type Examples Use

sequence typedef sequence <Shape, 100> All;

typedef sequence <Shape> All;

Bounded and unbounded sequences

of Shapes

Defines a type for a variable-length

sequence of elements of a specified IDL

type. An upper bound on the length may

be specified.

string string name;

typedef string<8> SmallString;

Unbounded and bounded sequences

of characters

Defines a sequence of characters,

terminated by the null character. An

upper bound on the length may be

specified.

array typedef octet uniqueId[12];

typedef GraphicalObject GO[10][8];

Defines a type for a multi-dimensional

fixed-length sequence of elements of a

specified IDL type.

record struct GraphicalObject {

string type;

Rectangle enclosing;

boolean isFilled;

};

Defines a type for a record containing a

group of related entities.

enumerated enum Rand

(Exp, Number, Name);

The enumerated type in IDL maps a type

name onto a small set of integer values.

union union Exp switch (Rand) {

case Exp: string vote;

case Number: long n;

case Name: string s;

};

The IDL discriminated union allows one

of a given set of types to be passed as an

argument. The header is parameterized

by an enum, which specifies which

member is in use.362 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

Inheritance: IDL interfaces may be extended through interface inheritance, as defined in

Section 8.2 above. For example, if interface B extends interface A, this means that it may

add new types, constants, exceptions, methods and attributes to those of A. An extended

interface can redefine types, constants and exceptions but is not allowed to redefine

methods. A value of an extended type is valid as the value of a parameter or result of the

parent type. For example, the type B is valid as the value of a parameter or result of the

type A. In addition, an IDL interface may extend more than one interface. For example,

interface Z here extends both B and C:

interface A { };

interface B: A{ };

interface C {};

interface Z : B, C {};

This means that Z has all of the components of both B and C (apart from those it

redefines), as well as those it defines as extensions.

When an interface such as Z extends more than one interface, there is a possibility

that it may inherit a type, constant or exception with the same name from two different

interfaces. For example, suppose that both B and C define a type called Q; the use of Q

in the Z interface will be ambiguous unless a scoped name such as B::Q or C::Q is given.

IDL does not permit an interface to inherit methods or attributes with common names

from two different interfaces.

All IDL interfaces inherit from the type Object, which implies that all IDL

interfaces are compatible with the type Object – which includes remote object

references. This makes it possible to define IDL operations that can takes as an argument

or return as a result a remote object reference of any type. The bind and resolve

operations in the Naming service are examples.

IDL type identifiers: As we will see in section 8.3.2, type identifiers are generated by the

IDL compiler for each type in an IDL interface. For example, the IDL type for the

interface Shape (Figure 8.3) might be:

IDL:Whiteboard/Shape:1.0

This example shows that an IDL type name has three parts – the IDL prefix, a type name

and a version number. Since interface identifiers are used as keys for accessing interface

definitions in the interface repository (described in Section 8.3.2), programmers must

ensure that they provide a unique mapping to the interfaces themselves. Programmers

may use the IDL prefix pragma to prefix an additional string to the type name in order

to distinguish their own types from those of others.

IDL pragma directives: These allow additional, non-IDL properties to be specified for

components in an IDL interface (see Henning and Vinoski [1999]). These properties

include, for example, specifying that an interface will be used only locally, or supplying

the value of an interface repository ID. Each pragma is introduced by #pragma and

specifies its type, for example:

#pragma version Whiteboard 2.3

CORBA language mappings • The mapping from the types in IDL to types in a given

programming language is quite straightforward. For example, in Java the primitive types

in IDL are mapped to the corresponding primitive types in that language. Structs, enumsSECTION 8.3 CASE STUDY: CORBA 363

and unions are mapped to Java classes; sequences and arrays in IDL are mapped to

arrays in Java. An IDL exception is mapped to a Java class that provides instance

variables for the fields of the exception and constructors. The mappings in C++ are

similarly straightforward.

However, some difficulties arise with mapping the parameter-passing semantics

of IDL onto those of Java. In particular, IDL allows methods to return several separate

values via output parameters, whereas Java can have only a single result. The Holder

classes are provided to overcome this difference, but this requires the programmer to

make use of them, which is not altogether straightforward. For example, the method

getPerson in Figure 5.2 is defined in IDL as follows:

void getPerson(in string name, out Person p);

In Java, the equivalent method would be defined as:

void getPerson(String name, PersonHolder p);

and the client would have to provide an instance of PersonHolder as the argument of its

invocation. The holder class has an instance variable that holds the value of the argument

for the client to access by RMI when the invocation returns. It also has methods to

transmit the argument between server and client.

Although C++ implementations of CORBA can handle out and inout parameters

quite naturally, C++ programmers suffer from a different set of problems with

parameters, related to storage management. These difficulties arise when object

references and variable-length entities such as strings or sequences are passed as

arguments.

For example, in Orbix [Baker 1997] the ORB keeps reference counts to remote

objects and proxies and releases them when they are no longer needed. It provides

programmers with methods for releasing or duplicating them. Whenever a server

method has finished executing, the out arguments and results are released and the

programmer must duplicate them if they will still be needed. For example, a C++ servant

implementing the ShapeList interface will need to duplicate the references returned by

the method allShapes. Object references passed to clients must be released when they

are no longer needed. Similar rules apply to variable-length parameters.

In general, programmers using IDL not only have to learn the IDL notation itself

but also have an understanding of how its parameters are mapped onto the parameters

of the implementation language.

Asynchronous RMI • CORBA supports a form of asynchronous RMI that allows clients

to make non-blocking invocation requests on CORBA objects [OMG 2004e]. It is

intended to be implemented in the client. Therefore a server is generally unaware of

whether it is invoked synchronously or asynchronously. (One exception is the

Transaction Service, which does need to be aware of the difference.)

Asynchronous RMI adds two new variants to the invocation semantics of RMIs:

• callback, in which a client uses an extra parameter to pass a reference to a callback

with each invocation so that the server can call back with the results;

• polling, in which the server returns a valuetype object that can be used to poll or

wait for the reply.364 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

The architecture of asynchronous RMI allows an intermediate agent to be deployed to

make sure that the request is carried out and, if necessary, to store the reply. Thus it is

appropriate for use in environments where clients may become temporarily

disconnected – such as, for example, a client using a laptop on a train.

8.3.2 The architecture of CORBA

The CORBA architecture is designed to support the role of an object request broker that

enables clients to invoke methods in remote objects, where both clients and servers can

be implemented in a variety of programming languages. The main components of the

CORBA architecture are illustrated in Figure 8.5.

This figure should be compared with Figure 5.15, in which case it will be noted

that the CORBA architecture contains three additional components: the object adapter,

the implementation repository and the interface repository.

CORBA provides for both static and dynamic invocations. Static invocations are

used when the remote interface of the CORBA object is known at compile time,

enabling client stubs and server skeletons to be used. If the remote interface is not known

at compile time, dynamic invocation must be used. Most programmers prefer to use

static invocation because it provides a more natural programming model.

We now discuss the components of the architecture, leaving those concerned with

dynamic invocation until last.

ORB core • The role of the ORB core includes all the functionality of the

communication module of Figure 5.15. In addition, an ORB core provides an interface

that includes the following:

• operations enabling it to be started and stopped;

• operations to convert between remote object references and strings;

• operations to provide argument lists for requests using dynamic invocation.

Object adapter • The role of an object adapter is to bridge the gap between CORBA

objects with IDL interfaces and the programming language interfaces of the

Figure 8.5 The main components of the CORBA architecture

client server

proxy

or dynamic invocation

implementation

repository

object

adapter

ORB ORB

skeleton

or dynamic skeleton

client

program

interface

repository

Request

for A core Reply core

Servant

ASECTION 8.3 CASE STUDY: CORBA 365

corresponding servant classes. This role also includes that of the remote reference and

dispatcher modules in Figure 5.15. An object adapter has the following tasks:

• It creates remote object references for CORBA objects (see Section 8.3.3).

• It dispatches each RMI via a skeleton to the appropriate servant.

• It activates and deactivates servants.

An object adapter gives each CORBA object a unique object name, which forms part of

its remote object reference. The same name is used each time an object is activated. The

object name may be specified by the application program or generated by the object

adapter. Each CORBA object is registered with its object adapter, which keeps a remote

object table that maps the names of CORBA objects to their servants.

Each object adapter also has its own name, which forms part of the remote object

references of all of the CORBA objects it manages. This name may either be specified

by the application program or generated automatically.

Portable object adapter • The CORBA standard for object adapters is called the

Portable Object Adapter (POA). It is called portable because it allows applications and

servants to be run on ORBs produced by different developers [Vinoski 1998]. This is

achieved by means of the standardization of the skeleton classes and of the interactions

between the POA and the servants.

The POA supports CORBA objects with two different sorts of lifetimes:

• those whose lifetimes are restricted to that of the process in which their servants

are instantiated;

• those whose lifetimes can span the instantiations of servants in multiple processes.

The former have transient object references and the latter have persistent object

references (see Section 8.3.3).

The POA allows CORBA objects to be instantiated transparently; and in addition

it separates the creation of CORBA objects from the creation of th, servants that

implement those objects. Server applications such as databases with large numbers of

CORBA objects can create servants on demand, only when the objects are accessed. In

this case, they may use database keys for the object names, or they may use a single

servant to support all of these objects.

In addition, it is possible to specify policies to the POA, for example, as to whether

it should provide a separate thread for each invocation, whether the object references

should be persistent or transient and whether there should be a separate servant for each

CORBA object. The default is that a single servant can represent all of the CORBA

objects for its POA.

Note that implementations of CORBA provide interfaces to the functionality of

the POA and the ORB core through pseudo-objects, given this name because they

cannot be used like regular CORBA objects; for example, they cannot be passed as

arguments in RMIs. They do, though, have IDL interfaces and are implemented as

libraries. The POA pseudo-object includes, for example, one method for activating a

POAmanager and another method, servant\_to\_reference, for registering a CORBA

object; the ORB pseudo-object includes the method init, which must be called to

initialize the ORB, the method resolve\_initial\_references, which is used to find services366 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

such as the Naming service and the root POA, and other methods that enable

conversions between remote object references and strings.

Skeletons • Skeleton classes are generated in the language of the server by an IDL

compiler. As described in Section 5.4.2, remote method invocations are dispatched via

the appropriate skeleton to a particular servant, and the skeleton unmarshals the

arguments in request messages and marshals exceptions and results in reply messages.

Client stubs/proxies • These are in the client language. The class of a proxy (for objectoriented languages) or a set of stub procedures (for procedural languages) is generated

from an IDL interface by an IDL compiler for the client language. As before, the client

stubs/proxies marshal the arguments in invocation requests and unmarshal exceptions

and results in replies.

Implementation repository • An implementation repository is responsible for activating

registered servers on demand and for locating servers that are currently running. The

object adapter name is used to refer to servers when registering and activating them.

An implementation repository stores a mapping from the names of object adapters

to the pathnames of files containing object implementations. Object implementations

and object adapter names are generally registered with the implementation repository

when server programs are installed. When object implementations are activated in

servers, the hostname and port number of the server are added to the mapping:

Not all CORBA objects need to be activated on demand. Some objects, for example

callback objects created by clients, run once and cease to exist when they are no longer

needed. They do not use the implementation repository.

An implementation repository generally allows extra information to be stored

about each server, such as access control information as to who is allowed to activate it

or to invoke its operations. It is possible to replicate information in implementation

repositories in order to provide availability or fault tolerance.

Interface repository • The role of the interface repository is to provide information

about registered IDL interfaces to clients and servers that require it. For an interface of

a given type it can supply the names of the methods and, for each method, the names and

types of the arguments and exceptions. Thus, the interface repository adds a facility for

reflection to CORBA. Suppose that a client program receives a remote reference to a

new CORBA object. If the client has no proxy for it, it can ask the interface repository

about the methods of the object and the types of parameter each of them requires.

When an IDL compiler processes an interface, it assigns a type identifier to each

IDL type it encounters. For each interface registered with it, the interface repository

provides a mapping between the type identifier of that interface and the interface itself.

Thus, the type identifier of an interface is sometimes called the repository ID because it

may be used as a key to IDL interfaces registered in the interface repository.

Implementation repository entry:

object adapter name pathname of object

implementation

hostname and port number

of serverSECTION 8.3 CASE STUDY: CORBA 367

Every CORBA remote object reference includes a slot that contains the type

identifier of its interface, enabling clients that hold it to enquire its type of the interface

repository. Those applications that use static (ordinary) invocation with client proxies

and IDL skeletons do not require an interface repository. Not all ORBs provide an

interface repository.

Dynamic invocation interface • As suggested in Section 5.5, in some applications it

may be necessary to construct a client program without knowing all the proxy classes it

will need in the future. For example, an object browser might need to display

information about all the CORBA objects available in the various servers in a distributed

system. It is not feasible for such a program to include proxies for all of these objects,

particularly as new objects may be added to the system as time passes. CORBA does not

allow classes for proxies to be downloaded at runtime, as in Java RMI. The dynamic

invocation interface is CORBA’s alternative.

The dynamic invocation interface allows clients to make dynamic invocations on

remote CORBA objects. It is used when it is not practical to employ proxies. The client

can obtain from the interface repository the necessary information about the methods

available for a given CORBA object. The client may use this information to construct

an invocation with suitable arguments and send it to the server.

Dynamic skeletons • Again, as explained in Section 5.5, it may be necessary to add to

a server a CORBA object whose interface was unknown when the server was compiled.

If a server uses dynamic skeletons, then it can accept invocations on the interface of a

CORBA object for which it has no skeleton. When a dynamic skeleton receives an

invocation, it inspects the contents of the request to discover its target object, the method

to be invoked and the arguments. It then invokes the target.

Legacy code • The term legacy code refers to existing code that was not designed with

distributed objects in mind. A piece of legacy code may be made into a CORBA object

by defining an IDL interface for it and providing an implementation of an appropriate

object adapter and the necessary skeletons.

8.3.3 CORBA remote object references

CORBA specifies a format for remote object references that is suitable for use whether

or not the remote object is to be activated by an implementation repository. References

using this format are called interoperable object references (IORs). The following

figure is based on Henning [1998], which contains a more detailed account of IORs:

Stepping through the various fields:

• The first field of an IOR specifies the type identifier of the IDL interface of the

CORBA object. Note that if the ORB has an interface repository, this type name

IOR format

IDL interface type ID Protocol and address details Object key

interface repository

identifier or type

IIOP host domain

name

port number adapter name object name368 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

is also the interface repository identifier of the IDL interface, which allows the

IDL definition for the interface to be retrieved at runtime.

• The second field specifies the transport protocol and the details required by that

particular transport protocol to identify the server. In particular, the Internet InterORB protocol (IIOP) uses TCP, in which the server address consists of a host

domain name and a port number [OMG 2004a].

• The third field is used by the ORB to identify a CORBA object. It consists of the

name of an object adapter in the server and the object name of a CORBA object

specified by the object adapter.

Transient IORs for CORBA objects last only as long as the process that hosts those

objects, whereas persistent IORs last between activations of the CORBA objects. A

transient IOR contains the address details of the server hosting the CORBA object,

whereas a persistent IOR contains the address details of the implementation repository

with which it is registered. In both cases, the client ORB sends the request message to

the server whose address details are given in the IOR. Here is how the IOR is used to

locate the servant representing the CORBA object in the two cases:

Transient IORs: The server ORB core receives the request message containing the

object adapter name and object name of the target. It uses the object adapter name to

locate the object adapter, which uses the object name to locate the servant.

Persistent IORs: An implementation repository receives the request. It extracts the

object adapter name from the IOR in the request. Provided that the object adapter

name is in its table, it attempts if necessary to activate the CORBA object at the host

address specified in its table. Once the CORBA object has been activated, the

implementation repository returns its address details to the client ORB, which uses

them as the destination for RMI request messages, which include the object adapter

name and the object name. These enable the server ORB core to locate the object

adapter, which uses the object name to locate the servant, as before.

The second field of an IOR may be repeated so as to specify the host domain name and

port number of more than one destination, to allow for an object or an implementation

repository to be replicated at several different locations.

The reply message in the request-reply protocol includes header information that

enables the above procedure for persistent IORs to be carried out. In particular, it

includes a status entry that can indicate whether the request should be forwarded to a

different server, in which case the body of the reply includes an IOR that contains the

address of the server of the newly activated object.

8.3.4 CORBA services

CORBA includes specifications for services that may be required by distributed objects.

In particular, the Naming service is an essential addition to any ORB, as we will see in

our programming example in Section 8.3.5. An index to documentation on all of the

services can be found at OMG’s web site [www.omg.org]. Many of the CORBA

services are described in Orfali et al. [1996, 1997]. A summary of key CORBA Services

is included in Figure 8.6. Further details of such services can be found on the companion

web site [www.cdk5.net/corba].SECTION 8.3 CASE STUDY: CORBA 369

8.3.5 CORBA client and server example

This section outlines the steps necessary to produce client and server programs that use

the IDL Shape and ShapeList interfaces shown in Figure 8.2. This is followed by a

discussion of callbacks in CORBA. We use Java as the client and server languages, but

the approach is similar for other languages. The interface compiler idlj can be applied to

the CORBA interfaces to generate the following items:

• The equivalent Java interfaces – two per IDL interface. The name of the first Java

interface ends in Operations – this interface just defines the operations in the IDL

interface. The second Java interface has the same name as the IDL interface and

Figure 8.6 CORBA Services

CORBA Service Role Further details

Naming service Supports naming in CORBA, in particular mapping names to

remote object references within a given naming context (see

Chapter 9).

[OMG 2004b]

Trading service Whereas the Naming service allows objects to be located by

name, the Trading service allows them to be located by

attribute; that is, it is a directory service. The underlying

database manages a mapping of service types and associated

attributes onto remote object references.

[OMG 2000a,

Henning and

Vinoski 1999]

Event service Allows objects of interest to communicate notifications to

subscribers using ordinary CORBA remote method

invocations (see Chapter 6 for more on event services

generally).

[Farley 1998,

OMG 2004c]

Notification

service

Extends the event service with added capabilities including

the ability to define filters expressing events of interest and

also to define the reliability and ordering properties of the

underlying event channel.

[OMG 2004d]

Security service Supports a range of security mechanisms including

authentication, access control, secure communication,

auditing and nonrepudiation (see Chapter 11).

[Blakely 1999,

Baker 1997,

OMG 2002b]

Transaction

service

Supports the creation of both flat and nested transactions (as

defined in Chapters 16 and 17).

[OMG 2003]

Concurrency

control service

Uses locks to apply concurrency control to the access of

CORBA objects (may be used via the transaction service or as

an independent service).

[OMG 2000b]

Persistent state

service

Offers a persistent object store for CORBA, used to save and

restore the state of CORBA objects (implementations are

retrieved from the implementation repository).

[OMG 2002d]

Lifecycle service Defines conventions for creating, deleting, copying and

moving CORBA objects; for example, how to use factories to

create objects.

[OMG 2002e]370 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

implements the operations in the first interface as well as those in an interface

suitable for a CORBA object. For example, the IDL interface ShapeList results in

the two Java interfaces ShapeListOperations and ShapeList, as shown in Figure

8.7.

• The server skeletons for each idl interface. The names of skeleton classes end in

POA – for example, ShapeListPOA.

• The proxy classes or client stubs, one for each IDL interface. The names of these

classes end in Stub – for example, \_ShapeListStub.

• A Java class to correspond to each of the structs defined with the IDL interfaces.

In our example, classes Rectangle and GraphicalObject are generated. Each of

these classes contains a declaration of one instance variable for each field in the

corresponding struct and a pair of constructors, but no other methods.

• Classes called helpers and holders, one for each of the types defined in the IDL

interface. A helper class contains the narrow method, which is used to cast down

from a given object reference to the class to which it belongs, which is lower down

the class hierarchy. For example, the narrow method in ShapeHelper casts down to

class Shape. The holder classes deal with out and inout arguments, which cannot be

mapped directly onto Java. See Exercise 8.9 for an example of the use of holders.

Server program • The server program should contain implementations of one or more

IDL interfaces. For a server written in an object-oriented language such as Java or C++,

these interfaces are implemented as servant classes. CORBA objects are instances of

servant classes.

When a server creates an instance of a servant class, it must register it with the

POA, which makes the instance into a CORBA object and gives it a remote object

reference. Unless this is done, the CORBA object will not be able to receive remote

invocations. Readers who studied Chapter 5 carefully may realize that registering the

object with the POA causes it to be recorded in the CORBA equivalent of the remote

object table.

In our example, the server contains implementations of the interfaces Shape and

ShapeList in the form of two servant classes, together with a server class that contains

an initialization section (see Section 5.4.2) in its main method:

The servant classes: Each servant class extends the corresponding skeleton class and

implements the methods of an IDL interface using the method signatures defined in

Figure 8.7 Java interfaces generated by idlj from CORBA interface ShapeList

public interface ShapeListOperations {

Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException;

Shape[] allShapes();

int getVersion();

}

public interface ShapeList extends ShapeListOperations, org.omg.CORBA.Object,

org.omg.CORBA.portable.IDLEntity { }SECTION 8.3 CASE STUDY: CORBA 371

the equivalent Java interface. The servant class that implements the ShapeList

interface is named ShapeListServant, although any other name could have been

chosen. Its outline is shown in Figure 8.8. Consider the method newShape in line 1,

which is a factory method because it creates Shape objects. To make a Shape object

a CORBA object, it is registered with the POA by means of its servant\_to\_reference

method, as shown in line 2, which makes use of the reference to the root POA that

was passed on via the constructor when the servant was created. Complete versions

of the IDL interface and the client and server classes in this example are available at

www.cdk5.net/corba.

The server: The main method in the server class ShapeListServer is shown in Figure

8.9. It first creates and initializes the ORB (line 1), then gets a reference to the root

POA and activates the POAManager (lines 2 & 3). Then it creates an instance of

ShapeListServant, which is just a Java object (line 4), and in doing this passes on a

reference to the root POA. It then makes it into a CORBA object by registering it with

the POA (line 5). After this, it registers the server with the Naming service. It then

waits for incoming client requests (line 10).

Figure 8.8 ShapeListServant class of the Java server program for CORBA interface ShapeList

import org.omg.CORBA.\*;

import org.omg.PortableServer.POA;

class ShapeListServant extends ShapeListPOA {

private POA theRootpoa;

private Shape theList[];

private int version;

private static int n=0;

public ShapeListServant(POA rootpoa){

theRootpoa = rootpoa;

// initialize the other instance variables

}

public Shape newShape(GraphicalObject g) throws ShapeListPackage.FullException { 1

version++;

Shape s = null;

ShapeServant shapeRef = new ShapeServant( g, version);

try {

org.omg.CORBA.Object ref = theRootpoa.servant\_to\_reference(shapeRef); 2

s = ShapeHelper.narrow(ref);

} catch (Exception e) {}

if(n >=100) throw new ShapeListPackage.FullException();

theList[n++] = s;

return s;

}

public Shape[] allShapes(){ ... }

public int getVersion() { ... }

}372 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

A server using the Naming service first gets a root naming context using the

NamingContextHelper (line 6). The naming context defines the scope within which

a set of names apply (each of the names within a context must be unique). The server

then makes a NameComponent (line 7), with the NameComponent being an object

representing a name in CORBA. This has two parts, a name and a kind; the kind field

is purely descriptive (the field is used by applications and is not interpreted by the

Naming service). Names can be compound and represented by a path to the object in

the naming graph. In this example, compound naming is not used; rather, the path

consists of a single name as defined in line 8. Finally, the server uses the rebind

method of the Naming service (line 9), which registers the name and remote object

reference pair in the appropriate context. Clients carry out the same steps but use the

resolve method as shown in Figure 8.10, line 2.

The client program • An example client program is shown in Figure 8.10. It creates and

initializes an ORB (line 1), then contacts the Naming service to get a reference to the

remote ShapeList object by using its resolve method (line 2). After that it invokes its

allShapes method (line 3) to obtain a sequence of remote object references to all the

Shapes currently held at the server. It then invokes the getAllState method (line 4),

giving as its argument the first remote object reference in the sequence returned; the

result is supplied as an instance of the GraphicalObject class.

Figure 8.9 Java class ShapeListServer

import org.omg.CosNaming.\*;

import org.omg.CosNaming.NamingContextPackage.\*;

import org.omg.CORBA.\*;

import org.omg.PortableServer.\*;

public class ShapeListServer {

public static void main(String args[]) {

try{

ORB orb = ORB.init(args, null); 1

POA rootpoa = POAHelper.narrow(orb.resolve\_initial\_references("RootPOA")); 2

rootpoa.the\_POAManager().activate(); 3

ShapeListServant SLSRef = new ShapeListServant(rootpoa); 4

org.omg.CORBA.Object ref = rootpoa.servant\_to\_reference(SLSRef); 5

ShapeList SLRef = ShapeListHelper.narrow(ref);

org.omg.CORBA.Object objRef = orb.resolve\_initial\_references("NameService");

NamingContext ncRef = NamingContextHelper.narrow(objRef); 6

NameComponent nc = new NameComponent("ShapeList", ""); 7

NameComponent path[] = {nc}; 8

ncRef.rebind(path, SLRef); 9

orb.run(); 10

} catch (Exception e) { ... }

}

}SECTION 8.3 CASE STUDY: CORBA 373

The getAllState method seems to contradict our earlier statement that objects

cannot be passed by value in CORBA, because both client and server deal in instances

of the class GraphicalObject. However, there is no contradiction: the CORBA object

returns a struct, and clients using a different language might see it differently. For

example, in the C++ language the client would see it as a struct. Even in Java, the

generated class GraphicalObject is more like a struct because it has no methods.

Client programs should always catch CORBA SystemExceptions, which report on

errors due to distribution (see line 5). Client programs should also catch the exceptions

defined in the IDL interface, such as the FullException thrown by the newShape method.

This example illustrates the use of the narrow operation: the resolve operation of

the Naming service returns a value of type Object, and this type is narrowed to suit the

particular type required – (ShapeList).

Callbacks • Callbacks can be implemented in CORBA in a manner similar to the one

described for Java RMI in Section 5.5.1. For example, the WhiteboardCallback

interface may be defined as follows:

interface WhiteboardCallback {

oneway void callback(in int version);

};

This interface is implemented as a CORBA object by the client, enabling the server to

send the client a version number whenever new objects are added. But before the server

can do this, the client needs to inform the server of the remote object reference of its

Figure 8.10 Java client program for CORBA interfaces Shape and ShapeList

import org.omg.CosNaming.\*;

import org.omg.CosNaming.NamingContextPackage.\*;

import org.omg.CORBA.\*;

public class ShapeListClient{

public static void main(String args[]) {

try{

ORB orb = ORB.init(args, null); 1

org.omg.CORBA.Object objRef =

orb.resolve\_initial\_references("NameService");

NamingContext ncRef = NamingContextHelper.narrow(objRef);

NameComponent nc = new NameComponent("ShapeList", "");

NameComponent path [] = { nc };

ShapeList shapeListRef =

ShapeListHelper.narrow(ncRef.resolve(path)); 2

Shape[] sList = shapeListRef.allShapes(); 3

GraphicalObject g = sList[0].getAllState(); 4

} catch(org.omg.CORBA.SystemException e) {...} 5

}

}374 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

object. To make this possible, the ShapeList interface requires additional methods such

as register and deregister, as follows:

int register(in WhiteboardCallback callback);

void deregister(in int callbackId);

After a client has obtained a reference to the ShapeList object and created an instance of

WhiteboardCallback, it uses the register method of ShapeList to inform the server that

it is interested in receiving callbacks. The ShapeList object in the server is responsible

for keeping a list of interested clients and notifying all of them each time its version

number increases when a new object is added. The callback method is declared as

oneway so that the server may use asynchronous calls to avoid delay as it notifies each

client.

8.4 From objects to components

Distributed object middleware has been heavily deployed in a wide range of

applications, including the areas featured in Chapter 1: finance and commerce,

healthcare, education, transport and logistics and so on. The techniques incorporated in

CORBA and related platforms have proved to be successful in tackling many of the key

issues associated with distributed programming, especially related to resolving

heterogeneity and enabling portability and interoperability of distributed systems

software. The range of services that accompany such platforms also encourage the

development of software that is secure and reliable.

However, a number of shortcomings have been identified. This has led to the

emergence of what we shall define to be component-based approaches, as a natural

evolution from distributed object computing. This section charts this transition,

discussing the requirements that led to component-based approaches and provides a

definition of components before examining in more depth the styles of componentbased approaches adopted in distributed systems. This is followed by Section 8.5 which

presents two contrasting case studies of component technologies, Enterprise JavaBeans

and Fractal.

Issues with object-oriented middleware • As mentioned above, component-based

approaches emerged to tackle the problems identified with distributed object computing.

The problems were listed in Section 8.1 and are discussed in more detail below.

Implicit dependencies: A distributed object offers a contract to the outside world in terms

of the interface (or interfaces) it offers to the distributed environment. The contract

represents a binding agreement between the provider of the object and users of that

object in terms of its expected behaviour. It is often assumed that such interfaces provide

a complete contract for the deploying and use of this object. However, this is not the

case. The problem arises from the fact that the internal (encapsulated) behaviour of an

object is hidden. For example, an object may communicate with another object or

associated distributed system service through a remote method invocation or other

communication paradigm. If we look back at the CORBA server and client programs

shown in Figure 8.9 and Figure 8.10, respectively, we see that the server issues aSECTION 8.4 FROM OBJECTS TO COMPONENTS 375

callback to the client, but this is not apparent from the interface defined on the server.

Also, while both the client and the server communicate with the name service, again this

is not visible from the external view of that object (as offered by the interfaces).

More generally, a given object can make arbitrary calls to other application-level

objects or to distributed system services offering naming, persistence, concurrency

control, transactions, security and so on, and this is not captured in the external view of

the configuration. Implicit dependencies in the distributed configuration make it

difficult to ensure the safe composition of a configuration, to replace one object with

another, and for third-party developers to implement one particular element in a

distributed configuration.

Requirement: From this, there is a clear requirement to specify not only the

interfaces offered by an object but also the dependencies that object has on other

objects in the distributed configuration.

Interaction with the middleware: Despite the goals of transparency, it is clear that in using

distributed object middleware the programmer is exposed to many relatively low-level

details associated with the middleware architecture. Again, the client-server example

shown in Figures 8.9 and 8.10 provides an illustration of this. Despite this being a rather

simple application, there are many CORBA-related calls that are absolutely essential to

the operation of the application. These include calls associated with naming (as

mentioned above), with the POA and to the ORB core. In more complex examples, this

could include arbitrarily sophisticated code in terms of the creation and management of

object references, management of object lifecycles, activation and passivation policies,

management of persistent state and policies for mappings to underlying platform

resources such as threads. All of this can very quickly become a distraction from the

main purpose of the code, which is to create a particular application. This is all too

evident from the example cited above, where the actual code related to the whiteboard

application is minimal and interleaved with code related to distributed systems concerns.

Requirement: There is a clear need to simplify the programming of distributed

applications, to present a clean separation of concerns between code related to

operation in a middleware framework and code associated with the application, and

to allow the programmer to focus exclusively on the latter.

Lack of separation of distribution concerns: Programmers using distributed object

middleware also have to deal explicitly with non-functional concerns related to issues

such as security, transactions, coordination and replication. In technologies such as

CORBA and RMI, this is achieved by inserting appropriate calls to the associated

distributed system services within the objects. This has two repercussions:

• Programmers must have an intimate knowledge of the full details of all the

associated distributed system services.

• The implementation for a given object will contain application code alongside

calls to distributed system services and to the underlying middleware interfaces

(as mentioned above). The resultant tangling of concerns further increases the

complexity of distributed systems programming.376 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

Requirement: The separation of concerns alluded to above should extend also to

dealing with the full range of distributed system services, and the complexities of

dealing with such services should be hidden wherever possible from the programmer.

No support for deployment: While technologies such as CORBA and Java RMI make it

possible to develop arbitrary distributed configurations of objects, there is no support for

the deployment of such configurations. Rather, objects must be deployed manually on

individual machines. This can become a tiresome and error-prone process, particularly

with large-scale deployments consisting of many objects spread over a physical

architecture with a large number of (potentially heterogeneous) nodes. In addition to

physically placing the objects, objects must also be activated and appropriate bindings

created to other objects. Because of the lack of support for deployment, developers

inevitably resort to ad hoc strategies for deployment, which are then not portable to other

environments.

Requirement: Middleware platforms should provide intrinsic support for

deployment so that distributed software can be installed and deployed in the same

way as software for a single machine, with the complexities of deployment hidden

from the user.

These four requirements have led to the emergence of component-based approaches to

distributed systems development, alongside the emergence of component-based

middleware, including the style of middleware referred to as application servers.

Note that although component-based approaches have only gained importance in

recent years, their roots can be traced back to earlier projects addressing reconfiguration

in distributed systems (such as the Conic Project at Imperial College London [Magee

and Sloman 1989]).

Essence of components • For the purposes of this discussion, we adopt the definition of

components provided by Szyperski in his book on component software [Szyperski

2002]:

Components: A software component is a unit of composition with contractually

specified interfaces and explicit context dependencies only.

In this classic definition, the word ‘only’ refers to the fact that any context dependencies

must be explicit – that is, there are no implicit dependencies present.

Software components are like distributed objects in that they are encapsulated

units of composition, but a given component specifies both its interfaces provided to the

outside world and its dependencies on other components in the distributed environment.

The dependencies are also represented as interfaces. More specifically, a component is

specified in terms of a contract, which includes:

• a set of provided interfaces – that is, interfaces that the component offers as

services to other components;

• a set of required interfaces – that is, the dependencies that this component has in

terms of other components that must be present and connected to this component

for it to function correctly.

In a given component configuration, every required interface must be bound to a

provided interface of another component. This is also referred to as a softwareSECTION 8.4 FROM OBJECTS TO COMPONENTS 377

architecture consisting of components, interfaces and connections between interfaces.

We can see an example of such a configuration in Figure 8.11. This example shows the

architecture of a simple file system providing an interface to other users and in turn

requiring connection to a directory service component and a flat file service component.

The figure also shows additional connections to block and device modules, capturing the

overall architecture of this particular file system. (We will investigate the actual

architectures of distributed file systems in Chapter 12.)

Interfaces may be of different styles. In particular, many component-based

approaches offer two styles of interface:

• interfaces supporting remote method invocation, as in CORBA and Java RMI;

• interfaces supporting distributed events (as discussed in Chapter 6).

We will see examples of both styles of interface when we look at Enterprise JavaBeans

in Section 8.5.1.

Programming in component-based systems is concerned with the development of

components and their composition. The goal is to support a style of software

development that parallels hardware development in using off-the-shelf components

and composing them together to develop more sophisticated services: a move from

software development to software assembly. This approach therefore supports thirdparty development of software components and also makes it easier to adapt system

configurations at runtime, by replacing one component with another that is a precise

match in terms of provided and required interfaces.

Note that advocates of component-based approaches place significant emphasis

on this use of composition and see this as the cleanest approach to constructing complex

software systems. In particular, they advocate composition over inheritance, viewing

inheritance as creating additional forms of implicit dependency (this time between

Figure 8.11 An example software architecture

Block module

File service

Directory service

Flat file service

Device module

Required interface

Provided interface378 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

classes). This can lead to issues such as the fragile base class problem, where changes

to a base class may have unforeseen repercussions for other objects further down the

inheritance hierarchy [Szyperski 2002].

So far, we have addressed the first requirement highlighted above (in terms of

making all dependencies explicit), but not the next three, which refer to simplifying the

development and deployment of distributed applications. This further level of support

will become apparent when we examine how component-based approaches have

evolved in the distributed systems community.

Components and distributed systems • A range of component-based middleware

technologies have emerged, including Enterprise JavaBeans (discussed in Section 8.5.1

below) and the CORBA Component Model (CCM) [Wang et al. 2001], an evolution of

CORBA from an object-based to a component-based platform. Component-based

middleware builds on the philosophy captured above but also adds significant support

for distributed systems development and deployment, as discussed below.

Containers: The concept of containers is absolutely central to component-based

middleware. Containers support a common pattern often encountered in distributed

applications, which consists of:

• a front-end (perhaps web-based) client;

• a container holding one or more components that implement the application or

business logic;

• system services that manage the associated data in persistent storage.

(This is analogous to the three-tier model described in Section 2.3.2.)

The tasks of the container are to provide a managed server-side hosting

environment for components and to provide the necessary separation of concerns

alluded to above, where components deal with application concerns and the container

Figure 8.12 The structure of a container

Lifecycle interface

External (provided)

interfaces

Components

Calls to external

distributed system

services

Interception

Incoming invocationsSECTION 8.4 FROM OBJECTS TO COMPONENTS 379

deals with distributed systems and middleware issues, ensuring that non-functional

properties are achieved. The overall structure of a container is shown in Figure 8.12.

This shows a number of components encapsulated within a container; the container

does not provide direct access to the components but rather intercepts incoming

invocations and then takes appropriate actions to ensure the desired properties of the

distributed application are maintained. If we take the case of CORBA, for example, this

would include:

• managing the interaction with the underlying ORB core and POA functionality

and hiding this entirely from application developers;

• managing calls to appropriate distributed system services, including security and

transaction services, to provide the required non-functional properties of the

application, again transparently to the programmer.

Taken together, this can significantly simplify the development of distributed

applications, allowing the component developer to focus exclusively on applicationlevel concerns. For example, with a container approach, all the POA-related calls

featured in Figures 8.8 and 8.9 would be made by the container and not by the

component. Similarly, through the interception mechanism, the container can issue a

potentially complex sequence of calls to appropriate distributed system services to

implement the required non-functional properties. As an illustration of the latter point,

consider the implementation of a simple management policy to deal with concurrent

access to a component. This can be implemented in a manner that is transparent to the

component by intercepting the incoming invocation at the external interface, acquiring

a lock associated with the underlying component and then proceeding with the call on

the underlying operation on the component itself, ensuring the lock is released when the

invocation is completed (we will talk more about locks in Section 16.4, but for now a

general understanding will suffice).

Middleware that supports the container pattern and the separation of concerns

implied by this pattern is known as an application server. This style of distributed

programming is in widespread use in industry today. A wide range of application servers

are now available; a summary of the key approaches is presented in Figure 8.13. Section

8.5.1 examines the Enterprise JavaBeans specification as an example of an application

server.

Figure 8.13 Application servers

Technology Developed by Further details

WebSphere Application Server IBM [www.ibm.com]

Enterprise JavaBeans SUN [java.sun.com XII]

Spring Framework SpringSource

(a division of VMware)

[www.springsource.org]

JBoss JBoss Community [www.jboss.org]

CORBA Component Model OMG [Wang et al. 2001]

JOnAS OW2 Consortium [jonas.ow2.org]

GlassFish SUN [glassfish.dev.java.net]380 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

Support for deployment: Component-based middleware provides support for the

deployment of component configurations; releases of software are packaged as software

architectures (components and their interconnections) together with deployment

descriptors that fully describe how the configurations should be deployed in a

distributed environment.

Note that components are deployed into containers, and deployment descriptors

are interpreted by containers to establish the required policies for the underlying

middleware and distributed system services. A given container therefore includes a

number of components that require the same configuration in terms of distributed

system support.

Deployment descriptors are typically written in XML and include sufficient

information to ensure that:

• components are correctly connected using appropriate protocols and associated

middleware support;

• the underlying middleware and platform are configured to provide the right level

of support to the component configuration (for example, in CORBA, this would

include configuring the POA);

• the associated distributed system services are set up to provide the right level of

security, transaction support and so on.

Tools are also provided to interpret the deployment descriptors and ensure the correct

deployment in a given physical architecture.

8.5 Case studies: Enterprise JavaBeans and Fractal

The advantage of application servers is that they provide comprehensive support for one

style of distributed programming – the three-tier approach as explained above – and

most of the complexities associated with distributed programming are hidden from the

user. The disadvantages are that the approach is both prescriptive and heavyweight:

prescriptive in the sense that the approach mandates that particular style of systems

architecture and heavyweight in that application servers are large and complex software

systems that inevitably carry an overhead in terms of performance and resource

requirements. The approach works best on high-end server machines.

To counter this, a more stripped-down and minimal style of component

programming is also adopted in distributed systems. We refer to this style as lightweight

component models to distinguish them from the much more heavyweight application

server architectures. In this section, we present two case studies of component

technologies: Enterprise JavaBeans, a leading example of the application server

approach, and Fractal, an example of a lightweight component architecture.

8.5.1 Enterprise JavaBeans

Enterprise JavaBeans (EJB) [java.sun.com XII] is a specification of a server-side,

managed component architecture and a major element of the Java Platform, Enterprise

Edition (Java EE), a set of specifications for client-server programming. OtherSECTION 8.5 CASE STUDIES: ENTERPRISE JAVABEANS AND FRACTAL 381

specifications include Java RMI and JMS, as featured elsewhere in this book (in

Chapters 5 and 6, respectively).

EJB is defined as a server-side component model because it supports the

development of the classic style of application, where potentially large numbers of

clients interact with a number of services realized through components or configuration

of components. The components, which are known as beans in EJB, are intended to

capture the application (or business) logic, as defined in Chapter 2, with EJB also

supporting the separation between this application logic and its persistent storage in a

back-end database. In other words, EJB provides direct support for the three-tier

architecture introduced in Section 2.3.2.

EJB is managed in the sense that the container pattern introduced above (in

Section 8.4) is used to provide support for key distributed systems services including

transactions, security and lifecycle support. Typically, the container injects appropriate

calls to the associated services to provide the required properties, and the use of a

transaction manager or security services is completely hidden from the developer of the

associated beans (container-managed). It is also possible for the bean developer to take

more control over these operations (bean-managed).

The goal of EJB is to maintain a strong separation of concerns between the various

roles involved in developing distributed applications. The EJB specification identifies

the following key roles:

• the bean provider, who develops the application logic of the component(s);

• the application assembler, who assembles beans into application configurations;

• the deployer, who takes a given application assembly and ensures it can be

correctly deployed in a given operational environment;

• the service provider, who is a specialist in fundamental distributed system

services such as transaction management and establishes the desired level of

support in these areas;

• the persistence provider, who is a specialist in mapping persistent data to

underlying databases and in managing these relationships at runtime;

• the container provider, who builds on the above two roles and is responsible for

correctly configuring containers with the required level of distributed systems

support in terms of non-functional properties related to, for example, transactions

and security as well as the desired support for persistence;

• the system administrator, who is responsible for monitoring a deployment at

runtime and making any adjustments to ensure its correct operation.

Note that EJB is a heavyweight component architecture in the sense introduced above.

There is significant software complexity, particularly associated with the management

of containers. As such, the approach is prescriptive and intended for certain classes of

application only. As mentioned above, EJB is particularly suited to applications that

follow the three-tier architecture based on a back-end database accessed via a service

interface offered by the middle tier (the application logic). For example, this style of

architecture is common in many eCommerce applications where the database maintains

information on stock items, prices and availability, while the middle tier offers382 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

interfaces to browse the stock and purchase selected items. These are typically large and

complex systems that require support in terms of distributed system services, and hence

the overhead associated with container management is fully justified. We will use the

example of an eShop throughout this section as motivation and to illustrate the use of

EJB in this setting. Other classes of application will not follow this pattern, and hence

EJB is an inappropriate technology for such applications. Examples include peer-to-peer

structures that simply do not follow this tiered model and more lightweight applications

running on embedded devices where the overhead of EJB cannot be justified.

In this section, we focus on the features of EJB 3.0 [java.sun.com XII], released

in 2006. A very large number of implementations of EJB 3.0 are available both

commercially and from open source consortia. Leading examples include Spring, JBoss,

JOnAS and GlassFish.

The EJB component model • A bean in EJB is a component offering one or more

business interfaces to potential clients of that component, where interfaces can be either

remote, requiring the use of appropriate communication middleware (such as RMI or

JMS), or local, in which case more direct, and hence efficient, bindings are possible.

Relating back to the terminology introduced in Section 8.4, a business interface is

equivalent to a provided interface (we will see how EJB supports required interfaces

below, in the subsection on dependency injection). A given bean is represented by the

set of remote and local business interfaces together with an associated bean class that

implements the interfaces. Two main styles of bean are supported in the EJB 3.0

specification:

Session beans: A session bean is a component implementing a particular task within

the application logic of a service, for example to make a purchase in our eShop

application. A session bean persists for the duration of a service and maintains a

running conversation with the client for the duration of the session. Session beans can

be either stateful, maintaining associated conversational state (such as the current

status of the eCommerce transaction), or stateless, in which case no state is

maintained. Stateful session beans imply a conversation with a single client and

maintain the state of that conversation. In contrast, stateless beans can have many

concurrent conversations with different clients. The state associated with stateful

beans may or may not be persistent, as we discuss below.

Message-driven beans: Clients interact with session beans using local or remote

invocation. We have seen throughout this book that other communication paradigms

are also important for distributed systems development, including indirect

communication paradigms. The concept of a message-driven bean was introduced in

EJB 2.0 to support indirect communication and, in particular, the possibility to

interact with components using either message queues or topics, building directly on

the functionality offered by JMS (remember that both queues and topics are firstclass entities in JMS representing alternative intermediaries for messages – see

Section 6.4.3). In a message-drive bean, a business interface will be realized as a

listener-style interface reflecting the event-driven nature of the associated bean.

POJOs and annotations • The task of programming in EJB has been simplified

significantly through the use of Enterprise JavaBeanPOJOs (plain old Java objects)

together with Java Enterprise JavaBean annotations. A bean (that is the implementationSECTION 8.5 CASE STUDIES: ENTERPRISE JAVABEANS AND FRACTAL 383

of the bean’s business interfaces) is a plain old Java object: it consists of the application

logic written simply in Java with no other code relating to it being a bean. Annotations

are then used to ensure the correct behaviour in the EJB context. In other words, a bean

is a POJO supplemented by annotations.

Annotations were introduced in Java 1.5 as a mechanism for associating metadata

with packages, classes, methods, parameters and variables. This metadata can then be

used by frameworks to ensure the right behaviour or interpretation is associated with

that part of the program. As an example, annotations are used to introduce a bean of a

particular style. For example, the following are examples of annotated bean definitions

(representing the main styles of bean in EJB 3.0):

@Stateful public class eShop implements Orders {...}

@Stateless public class CalculatorBean implements Calculator {...}

@MessageDriven public class SharePrice implements MessageListener {...}

Annotations are also used to indicate whether business interfaces are remote (@Remote)

or local (@Local). The following example introduces the Orders interface as a remote

interface and the Calculator interface from the CalculatorBean as a local interface only:

@Remote public interface Orders {...}

@Local public interface Calculator {...}

As will become apparent, annotations are used throughout EJB, providing a

specification of how a program should be interpreted in an EJB context.

In the description that follows we will develop the eShop example as an

illustration of the extensive use of annotations in programming bean objects (in this cas,

a session bean).

Enterprise JavaBean containers in EJB • EJB adopts a container-based approach as

described in Section 8.4. Beans are deployed to containers, and the containers provide

implicit distributed system management using interception. In this way, the container

provides the necessary policies in areas including transaction management, security,

persistence and lifecycle management allowing the bean developer to focus exclusively

on the application logic. Containers must therefore be configured with the necessary

level of support. In the current version, EJB is preconfigured with common default

policies and the developer need only take action if these defaults are insufficient

(referred to as configuration by exception in the specification [java.sun.com XII]).

A significant number of annotations are defined to control the various aspects

mentioned above. We illustrate their use by focusing on Enterprise JavaBean transaction

management and encourage the reader to also look at the EJB 3.0 specifications for

further examples. Transactions will be introduced in Chapters 16 and 17. In outline,

though, their role is to ensure that all objects (or, in this context, components) managed

by a single server (or multiple servers in the case of distributed transactions) remain in

a consistent state in spite of concurrent access from multiple clients and in the event of

server failure. They achieve this by enabling a sequence of operations to be executed

atomically, in that the sequence of operations either completes successfully in a manner

that is free from interference from other concurrent access, or in the presence of a failure

(such as, a server crash), has no effect at all Returning to our eShop example, a

transaction mechanism will ensure, for example, that two concurrent purchases do not

result in a single item being sold twice and that a server crash does not allow the system384 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

to get into an inconsistent state where an item has been paid for but not assigned to the

purchaser.

The mechanisms to achieve transactions are relatively complex, hence the two

chapters devoted to this area later in the book. Nevertheless, the overall concept is

relatively straightforward, and an intuitive understanding will suffice to understand how

EJB manages transactions. The key thing to bear in mind is that transactions refer to

sequences of operations, and that the sequences must be clearly identified for the

transaction management service to do its job. Transactions in EJB apply equally to any

style of bean, including both session beans and message-driven beans.

The first thing to declare is whether transactions associated with an enterprise

bean should be bean-managed or container-managed. This is achieved by associating the

following annotations with the associated class, respectively:

@TransactionManagement (BEAN)

@TransactionManagement (CONTAINER)

Bean-managed transactions are the most straightforward to understand. In this case, the

bean developer is responsible for explicitly identifying the sequence of operations to be

included within the transaction. This is achieved by explicitly including two methods

from the Java interface javax.transaction.UserTransaction – the User.Transaction.begin

and UserTransaction.commit methods – within the code of the bean. These can be used

at either the client or the server end of an interaction. The following code fragment

illustrates the use of this bean-managed approach in the eShop example:

@Stateful

@TransactionManagement (BEAN)

public class eShop implements Orders {

@Resource javax.transaction.UserTransaction ut;

public void MakeOrder (...) {

ut.begin ();

...

ut.commit ();

}

}

To a certain extent, however, this is against the spirit of the container approach as it

requires the inclusion of transaction-related code within the bean. The alternative,

container-managed transaction, obviates the need for this explicit code by allowing the

container to determine when to start and finish a transaction. This is achieved through

the association of a given demarcation policy with the bean execution. Again, this is

achieved declaratively by associating an appropriate annotation with a given method

within the bean class. For example, consider the following code fragment:

@Stateful public class eShop implements Orders {

...

@TransactionAttribute (REQUIRED)

public void MakeOrder (...) {

...

}

}SECTION 8.5 CASE STUDIES: ENTERPRISE JAVABEANS AND FRACTAL 385

This shows the association of the REQUIRED policy with the MakeOrder method. This

policy sates that the associated method must be carried out within a transaction. To

understand this policy, it is necessary to realize that a transaction may be initiated by the

caller or may be the responsibility of the bean itself. The REQUIRED policy starts a new

transaction if necessary – that is, if the caller does not provide a transaction context

indicating the work is already being carried out within a transaction. This and other

policies are summarized in Figure 8.14.

Note that, by default, transactions are container-managed in EJB.

Dependency injection: The example above also illustrates a further important role of

containers: Enterprise JavaBean dependency injection. Dependency injection is a

common pattern in programming whereby a third party, in this case a container, is

responsible for managing and resolving the relationships between a component and its

dependencies (the required interfaces, in the terminology of Section 8.4). In particular,

in EJB 3.0, a component refers to a dependency using an annotation and the container is

responsible for resolving this annotation and ensuring that, at runtime, the associated

attribute refers to the right object. This is typically implemented by the container using

reflection.

For example, in the code fragment above, the @Resource annotation indicates a

dependency of this component on an object implementing the UserTransaction

interface. This simply must exist for the configuration to make sense. Dependency

injection both flags this dependency and ensures that, when the correct component

configuration is deployed, the associated attribute ut refers to the right external resource.

Figure 8.14 Transaction attributes in EJB.

Attribute Policy

REQUIRED If the client has an associated transaction running, execute

within this transaction; otherwise, start a new transaction.

REQUIRES\_NEW Always start a new transaction for this invocation.

SUPPORTS If the client has an associated transaction, execute the

method within the context of this transaction; if not, the call

proceeds without any transaction support.

NOT\_SUPPORTED If the client calls the method from within a transaction, then

this transaction is suspended before calling the method and

resumed afterwards – that is, the invoked method is excluded

from the transaction.

MANDATORY The associated method must be called from within a client

transaction; if not, an exception is thrown.

NEVER The associated methods must not be called from within a

client transaction; if this is attempted, an exception is

thrown.386 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

Enterprise JavaBean Interception • The Enterprise JavaBeans specification enables the

programmer to intercept two types of operation on beans in order to alter their default

behaviour:

• method calls associated with a business interface;

• lifecycle events.

We look at each in turn below.

Interception of methods: This mechanism is used where it is necessary to associate a

particular action or set of actions with an incoming call on a business interface. This

applies equally to incoming invocations on a session bean or incoming events on a

message-driven bean. As we have already seen, interception is used widely in the EJB

architecture to provide implicit management. This allows the application developer to

extend the use of interception to more domain-specific concerns not provided by the

container.

Consider the running example of the eShop. Suppose there is a need within the

eShop to implement logging of all operations carried out in the system, for example for

auditing purposes. Interception allows the programmer to introduce such a service

without changing the application logic contained in the bean. As a second example, the

interception mechanism could be used to prevent certain customers from making

purchases in the eShop (for example, if they have defaulted on previous payments).

There are several ways of associating interceptors with a given bean, including

associating an interception class with a given bean class or individual method (using the

annotation @Interceptors), or associating an interception method with a given class

(using the annotation @AroundInvoke). For the sake of simplicity, we focus on the latter

mechanism and return to our example of an eShop:

@Stateful

public class eShop implements Orders {

public void MakeOrder (...) {

...

}

@AroundInvoke

public Object log(InvocationContext ctx) throws Exception {

System.out.println ("The following method was invoked: " +

ctx.getMethod().getName());

return invocationContext.proceed();

}

}

The annotation @AroundInvoke introduces an interceptor on the eShop bean class. The

interceptor method must have the following syntax:

Object <methodName>(javax.ejb.InvocationContext)

This method is then called whenever any of the business methods are called on eShop.

The associated parameter adds significantly to the capabilities of interceptors by

providing both metadata associated with the invocation being intercepted (for example,

references to the bean, the method invoked and the actual parameters associated with theSECTION 8.5 CASE STUDIES: ENTERPRISE JAVABEANS AND FRACTAL 387

invocation) and also limited capabilities to intercede – that is, to change the parameters

before the method is executed. The last line of the method, the call to proceed, returns

control back to the intercepted method (or to the next interceptor in the chain if more

than one interceptor is defined).

The main methods associated with the invocation context are summarized in

Figure 8.15.

Interception of lifecycle events: A similar mechanism can be used to intercept and react to

lifecycle events associated with a component. In particular, the EJB specification allows

a bean developer to associate interceptors with the creation and deletion of components

using the following annotations, respectively:

@PostConstruct

@PreDestroy

The annotations are associated with given methods in the bean class, with the effect that

these methods will be called when the associated lifecycle events happen. For example,

in the code fragment below from the eShop, TidyUp will be called just before the

component is destroyed:

@Stateful

public class eShop implements Orders {

...

public void MakeOrder (...) { ...}

...

@PreDestroy void TidyUp() { ... }}

}

This annotation is generally used to release any resources currently in use by the eShop

class, for example, open files or sockets associated with the eShop implementation.

Figure 8.15 Invocation contexts in EJB

Signature Use

public Object getTarget() Returns the bean instance associated with the

incoming invocation or event

public Method getMethod() Returns the method being invoked

public Object[] getParameters() Returns the set of parameters associated with the

intercepted business method

public void setParameters(

Object[] params)

Allows the parameter set to be altered by the

interceptor, assuming type correctness is

maintained

public Object proceed() throws

Exception

Execution proceeds to next interceptor in the chain

(if any) or the method that has been intercepted388 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

Similarly, TidyUp could be used to ensure key data associated with the eShop is written

to the back-end database.

If the bean is stateful, it is also possible to capture activation and passivation

events using @PostActivate and @PrePassivate. Again, this allows key actions to be

taken in association with these lifecycle events – for example, ensuring conversational

state associated with a session is stored in the database before the bean is passivated.

Annotations are used throughout EJB 3.0 to provide a consistent and simple

programming model whereby component developers construct application logic in

POJOs and then decorate this, where appropriate, with additional meta-level annotations

that are interpreted by the container framework.

8.5.2 Fractal

As mentioned above, Fractal is a lightweight component model that brings the benefits

of component-based programming to the development of distributed systems [Bruneton

et al. 2006, fractal.ow2.org I]. Fractal provides support for programming with

interfaces, with the associated benefits in terms of the separation of interface and

implementation (benefits also provided by distributed objects). Fractal goes further,

though, and supports the explicit representation of the software architecture of the

system, avoiding the problem of implicit dependencies discussed in Section 8.4. The

approach is deliberately minimal, with no support for additional component-related

functionality such as deployment, the full container pattern or the enriched

programming model offered by application servers. Fractal is used to construct more

complex software systems (including middleware systems as discussed below) using the

component model as the basic building block, resulting in software that has a clear

component-based architecture and that is configurable and also reconfigurable at

runtime to match the current operational environment and requirements.

Fractal defines a programming model and, as such, is programming language–

agnostic. Implementations of this model are available in several different languages,

including:

• Julia and AOKell (Java-based, with the latter also offering support for aspectoriented programming);

• Cecilia and Think (C-based);

• FracNet (.NET-based);

• FracTalk (Smalltalk-based);

• Julio (Python-based).

Julia and Cecilia are treated as the reference implementations of Fractal.

Fractal is supported by the OW2 consortium [www.ow2.org], an open source

software community for distributed systems middleware that encourages and promotes

the component-based philosophy for the construction of such software. To date, Fractal

has been used in the construction of a wide range of middleware platforms including

Think (a configurable operating system kernel), DREAM (a middleware platform

supporting various forms of indirect communication), Jasmine (a tool supporting the

monitoring and management of SOA platforms), GOTM (offering flexible transactionSECTION 8.5 CASE STUDIES: ENTERPRISE JAVABEANS AND FRACTAL 389

management) and Proactive (a middleware platform for Grid computing). Fractal is also

the basis of the Grid Component Model (GCM), which has been influential in the

development of associated ETSI standards [Baude et al. 2009]. Further details of all

these projects can be found on the OW2 web site [www.ow2.org].

Note that a some other lightweight component models have been developed

specifically for distributed systems. We feature two – OpenCOM and OSGI – in the box

on the next page.

The core component model • A component in Fractal offers one or more interfaces,

with two types of interfaces available:

• server interfaces, which support incoming operational invocations (equivalent to

provided interfaces in the terminology of Section 8.4);

• client interfaces, which support outgoing invocations (equivalent to required

interfaces).

An interface is an implementation of an interface type, which defines the operations that

are supported by that interface.

Bindings in Fractal: To enable composition, Fractal supports bindings between

interfaces. Two styles of binding are supported by the model:

Primitive bindings: The simplest style of binding is a primitive binding, which is a

direct mapping between one client interface and one server interface within the same

address space, assuming the types are compatible. Primitive bindings can be

implemented efficiently in a given language environment, for example through direct

object references.

Composite bindings: Fractal also supports composite bindings, which are arbitrarily

complex software architectures (that is consisting of components and bindings)

implementing communication between a number of interfaces potentially on

different machines. For example, if you were implementing a CORBA connection in

Fractal, the binding would be composed of components representing the core

architectural elements in CORBA, including proxies, the ORB core, object adapters,

skeletons and servants (mirroring the architecture in Figure 8.5).

Composite bindings are themselves components in Fractal, and this is important

for two reasons:

• A system developed using Fractal is fully configurable in terms of the components

and their interconnections. For example, a configuration can be established

wherein components interact using a composite binding implementing any of the

communication paradigms discussed in Chapters 5 and 6 (remote invocation or

indirect, point-to-point or multiparty, and so on). If a given communication

paradigm is not already provided, it can be developed in Fractal and then made

available to future developers as a component.

• Once established, any aspect of the software architecture can be reconfigured at

runtime, including composite bindings. It is very useful to be able to adapt

communication structures at runtime, for example to introduce added levels of

security or to alter the implementation to be more scalable as a system grows in

size.390 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

OpenCOM • OpenCOM [Coulson et al. 2008] is a lightweight component model

with very similar goals to Fractal. OpenCOM is a minimal and open component

model that is designed to be domain– and operating environment–independent; that

is, the component technology is sufficiently flexible to be applied in any context

including demanding areas such as resource-limited wireless sensor networks.

OpenCOM is also designed to offer negligible overhead in terms of performance and

memory requirements, allowing it to be used in situations where performance is

critical – for example, in router implementations.

The overall architecture of OpenCOM consists of a minimal runtime kernel

supporting basic component operations including loading and unloading a

component, and binding components together. This is then enhanced with optional

reflective and platform extensions, which support the dynamic loading of reflective

capabilities and also different models underpinning key platform operations,

including the semantics of loading and binding. The extensions are therefore

conceptually similar to controllers in Fractal.

OpenCOM has been used in the development of a variety of experimental

middleware platforms, including ReMMoC [Grace et al. 2003], which offers service

discovery in highly heterogeneous ubiquitous computing environments (see Chapter

19), and GridKIT [Grace et al. 2008], an experimental, highly configurable and

reconfigurable middleware framework for Grid computing, which also features an

open overlays framework for the construction of arbitrary network virtualizations

including structured and unstructured peer-to-peer overlays.

OSGi • OSGi [www.osgi.org] is a specification of a Java-based middleware

platform managed by the open standards organization, the OSGi Alliance. A number

of implementations of this specification exist, including Equinox, Knopflerfish, Felix

and Concierge. The platform supports the deployment and subsequent lifecycle

management and adaptation of modular software systems organized as

communicating bundles (similar to components). Bundles communicate through one

or more service interfaces with services published in a service registry, thus

supporting dynamic binding. As the unit of lifecycle management, a given bundle can

be installed, started, activated, stopped and uninstalled. Bundles can also be

dynamically deployed at runtime, and existing bundles updated. OSGi was originally

developed for the programming of service gateways (hence the original name of

Open Service Gateway i) but is now used in a wide variety of application domains,

including in the programming of mobile phones, as middleware for Grid computing

and also as the plug-in architecture in the Eclipse Integrated Development

Environment (IDE), a popular multi-language framework for software development.

OSGi targets the deployment and management of centralized configurations of

software, for example residing on a single device or on a server. A distributed

implementation of OSGI, R-OSGi, has also been developed [Rellermeyer et al.

2007]. This allows software architectures to be distributed at service boundaries

across arbitrary networked configurations. R-OSGi uses the proxy pattern, first

introduced in Chapter 2, to obtain transparent distribution at such boundaries.SECTION 8.5 CASE STUDIES: ENTERPRISE JAVABEANS AND FRACTAL 391

We look in more detail at support for reconfiguration in the subsection on membranes

and controllers below.

Hierarchical composition: The component model is hierarchical in that a component

consists of a series of subcomponents and associated bindings, where the

subcomponents may themselves be composite. For example, the ORB core in the above

example could be further decomposed, given its inherent complexity. Composition is

supported by a Fractal Architectural Description Language (ADL), which we introduce

by a simple example showing the creation of a component containing two

subcomponents that interact in a client-server manner:

<definition name="cs.ClientServer">

<interface name="r" role="server"

signature="java.lang.Runnable" />

<component name="caller" definition="hw.CallerImpl" />

<component name="callee" definition="hw.CalleeImpl" />

<binding client="this.r" server="caller.r" />

<binding client="caller.s" server="callee.s" />

</definition>

Fractal ADL is based on XML. This example shows a component cs.ClientServer with

two subcomponents, caller and callee; bindings are created between the client interface,

this.r (that is, the r interface defined on the containing component cs.ClientServer), and

the associated caller.r interface (the r interface defined on the caller component), and

between the client interface caller.s and the corresponding server interface callee.s. The

associated configuration is illustrated in Figure 8.16.

Fractal also supports sharing, whereby a given component may be shared across

multiple software architectures. The developers of Fractal argue that this is necessary to

faithfully represent system architectures including access to underlying resources that

are fundamentally shared, such as a TCP connection. As a further example, it would be

possible for the callee (the server component) to be shared across multiple

configurations.

Membranes and controllers • In implementation, a component consists of a membrane,

which defines control capabilities associated with the component through a set of

controllers, and also the associated content – the subcomponents (and bindings) that

make up its architecture. Interfaces can be internal to the membrane and hence only

Figure 8.16 An example component configuration in Fractal

Caller Callee

r

r

s s

cs.ClientServer392 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

visible to components within the content part, or external and hence visible to any other

components. This structure is illustrated in Figure 8.17.

The membrane concept is crucial to the Fractal approach; a membrane provides a

configurable control regime for the encapsulated set of components (the content). In

other words, the set of controllers defines the control capabilities and associated

semantics for these components. By changing the set of controllers, wealso change the

capabilities.

Controllers can be used for various purposes:

• One of the key uses of controllers is to implement lifecycle management,

including operations associated with activation and passivation such as suspend,

resume and checkpoint. For example, Fractal supports a LifeCycleController

supporting three methods, startFc, stopFc and getFcState, which implements

these three functions, respectively. This is crucial in cases where reconfigurations

of the underlying software architecture are being carried out at runtime. Consider

the simple example of the client-server configuration above and suppose the

server is to be replaced dynamically by an enhanced server (perhaps one

supporting multi-threading for improved throughput). In this case, to avoid

inconsistencies, it is helpful to suspend the configuration, replace the callee

component with the new one, and then resume the configuration.

• Controllers also offer reflection capabilities (see Chapter 2). In particular,

introspection capabilities are provided through two interfaces, Component and

ContentController, which support introspection (dynamic discovery) of the

interfaces associated with a component and step through the architecture of a

composite component structure respectively. The full interfaces for the two

controllers are shown in Figure 8.18. Introspection is again important to support

dynamic reconfiguration. Returning to our client-server example, it is possible,

through the above interfaces, to discover the precise architecture of the underlying

component configuration (in this case, a simple configuration consisting of two

Figure 8.17 The structure of a Fractal component

Control interfaces

Membrane

Client interface

Server interfaces

Content

ControllersSECTION 8.5 CASE STUDIES: ENTERPRISE JAVABEANS AND FRACTAL 393

components) and also to ensure that a replacement callee component supports

precisely the same interface as the old one.

• Controllers can be introduced to offer interception capabilities mirroring the

capability offered in EJB and reported in Section 8.5.1 above. Interception is a

powerful mechanism with many uses. In the EJB section, the example is provided

of using interception to implement logging. For example, interception could be

used in the client-server example to log all calls issued by the caller component in

a manner that would be completely transparent to both the caller and callee. A

further use of interception is to implement an access control policy only allowing

an invocation to proceed if a given principal has rights to access a given resource

(see Section 11.2.4).

Having studied the relative roles of membranes and controllers, it is now possible to

relate membranes to containers, as introduced in Section 8.4 and in the EJB case study

above. Membranes, like containers, provide a place for the deployment of components;

both techniques also support implicit distributed systems management, containers by

making implicit calls to distributed systems services and membranes through their

constituent controllers. Membranes, though, are significantly more flexible:

• In terms of reflection, support can range from black-box components where

internal structure is hidden, through approaches where limited introspection

capabilities are offered (dynamically discovering interfaces), to advanced

reflection features supporting full introspection and providing inherent support for

subsequent adaptation of internal structures.

• In terms of supporting non-functional concerns, at one extreme, membranes can

provide no more than a simple encapsulation of components (if, for example,

minimal controllers are deployed); at the other extreme, they can support fully

fledged distributed systems management of components, including support for

transactions and security as in application servers, but in a completely

configurable and reconfigurable manner.

Figure 8.18 Component and ContentController Interfaces in Fractal

public interface Component {

Object[] getFcInterfaces ();

Object getFcInterface (String itfName);

Type getFcType ();

}

public interface ContentController {

Object[] getFcInternalInterfaces ();

Object getFcInterfaceInterface(String itfName);

Component[] getFcSubComponents ();

void addFcSubComponent (Component c);

void removeFcSubComponent(Component c);

}394 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

Because of this inherent flexibility, Fractal is referred to as a open component model.

A worked example of a complete Fractal implementation can be found in the

online tutorial [fractal.ow2.org II]. This example shows how Fractal can be used to

implement a configurable, minimal HTTP server called Comanche.

8.6 Summary

This chapter has examined the design of complete middleware solutions based around

distributed objects and components. As will now be apparent, they represent a natural

evolution in thinking about such programming abstractions. Distributed objects are

important in terms of bringing the benefits of encapsulation and data abstraction to

distributed systems, and as well as associated tools and techniques from the field of

object-oriented design. Distributed objects therefore represent a significant step forward

from previous approaches based directly on the client-server model. In applying the

distributed object approach, however, a number of significant limitations have emerged

and these have been presented and analyzed in this chapter. In summary, it is often too

complex in practice to use middleware solutions such as CORBA for sophisticated

distributed applications and services, particularly when dealing with advanced

properties of such systems such as, for example, dependability (fault tolerance and

security).

Component technologies overcome these limitations, through their intrinsic

separation of concerns between application logic and distributed systems management.

The explicit identification of dependencies also helps in terms of supporting third party

composition of distributed systems. This chapter examined the EJB 3.0 specification

which has made further steps forward in terms of simplifying distributed systems

development through an approach that emphasizes the use of plain old Java objects with

the complexities managed declaratively through the use of Java annotations. As we saw,

more lightweight technologies such as Fractal and OpenCOM have also been introduced

to bring the benefits of component-based programming to the development of

middleware platforms themselves, with little added overhead in terms of performance.

Component technologies are important for the development of distributed

applications, but like any technology, they have their strengths and weaknesses. For

example, the component approach is quite prescriptive and best suited to applications

that naturally resemble three-tier architectures. To offer a broader perspective on the

range of available middleware platforms, the next two chapters examine alternative

approaches based on the adoption of web-based standards (web services) and peer-topeer systems.EXERCISES 395

EXERCISES

8.1 The Task Bag is an object that stores (key, value) pairs . A key is a string and a value is

a sequence of bytes. Its interface provides the following remote methods:

pairOut, with two parameters through which the client specifies a key and a value to

be stored.

pairIn, whose first parameter allows the client to specify the key of a pair to be

removed from the Task Bag. The value in the pair is supplied to the client via a

second parameter. If no matching pair is available, an exception is thrown.

readPair, which is the same as pairIn except that the pair remains in the Task Bag.

Use CORBA IDL to define the interface of the Task Bag. Define an exception that can

be thrown whenever any one of the operations cannot be carried out. Your exception

should return an integer indicating the problem number and a string describing the

problem. The Task Bag interface should define a single attribute giving the number of

tasks in the bag. page 357

8.2 Define an alternative signature for the methods pairIn and readPair, whose return value

indicates when no matching pair is available. The return value should be defined as an

enumerated type whose values can be ok and wait. Discuss the relative merits of the two

alternative approaches. Which approach would you use to indicate an error such as a key

that contains illegal characters? page 358

8.3 Which of the methods in the Task Bag interface could have been defined as a oneway

operation? Give a general rule regarding the parameters and exceptions of oneway

methods. In what way does the meaning of the oneway keyword differ from the

remainder of IDL? page 358

8.4 CORBA supports passing non-CORBA objects by value. What are the properties of

these non-CORBA objects? What are their limitations? page 360

8.5 In Figure 8.2 the type All was defined as a sequence of fixed length. Redefine this as an

array of the same length. Give some recommendations as to the choice between arrays

and sequences in an IDL interface. page 361

8.6 The Task Bag is intended to be used by cooperating clients, some of which add pairs

(describing tasks) and others of which remove them (and carry out the tasks described).

When a client is informed that no matching pair is available, it cannot continue with its

work until a pair becomes available. Define an appropriate callback interface for use in

this situation. page 373

8.7 What are CORBA security services used for? page 369

8.8 The grammar of IDL is a subset of ANSI C++ with additional constructs to support

method signatures. Give the uses of CORBA IDL modules and interfaces. page 358396 CHAPTER 8 DISTRIBUTED OBJECTS AND COMPONENTS

8.9 Use the Java IDL compiler to process the interface you defined in Exercise 8.1. Inspect

the definition of the signatures for the methods pairIn and readPair in the generated

Java equivalent of the IDL interface. Look also at the generated definition of the holder

method for the value argument for the methods pairIn and readPair. Now give an

example showing how the client will invoke the pairIn method, explaining how it will

acquire the value returned via the second argument. page 362

8.10 What are the new variants to the invocation semantics used by asynchronous RMI?

page 363

8.11 “A software component is a unit of composition with contractually specified interfaces

and explicit context dependencies only.” What is the significance of the word ‘only’ in

this statement? page 376

8.12 Containers support a common pattern often encountered in distributed applications.

What does this pattern consist of? page 378

8.13 What are the key roles of the EJB specification? Briefly explain each.

page 381

8.14 What are the transaction attributes in EJB?

page 385

8.15 Explain carefully how component-based middleware in general and EJB in particular

can overcome the key limitations of distributed object middleware. Provide examples to

illustrate your answer. page 374-380

8.16 Discuss whether the EJB architecture would be suitable to implement a massively

multiplayer online game (an application domain initially introduced in Section 1.2.2).

What would be the strengths and weaknesses of using EJB in this domain? page 380

8.17 What are the main methods associated with the invocation context? page 387

8.18 Fractal supports bindings between interfaces. What are the two styles of binding

supported by the model? page 389

8.19 What is the use of the Fractal interface OpenCOM? page 390

8.20 Fractal defines a programming model and, as such, is programming language-agnostic.

Name the languages in which the implementations of this model are available.

page 388397

9

WEB SERVICES

9.1 Introduction

9.2 Web services

9.3 Service descriptions and IDL for web services

9.4 A directory service for use with web services

9.5 XML security

9.6 Coordination of web services

9.7 Applications of web services

9.8 Summary

A web service provides a service interface enabling clients to interact with servers in a

more general way than web browsers do. Clients access the operations in the interface of

a web service by means of requests and replies formatted in XML and usually transmitted

over HTTP. Web services can be accessed in a more ad hoc manner than CORBA-based

services, enabling them to be more easily used in Internet-wide applications.

As with CORBA and Java, the interfaces of web services can be described in an IDL.

But for web services, additional information, including the encoding and communication

protocols in use and the service locations must be provided.

Users require a secure means for creating, storing and modifying documents and

exchanging them over the Internet. The secure channels of Transport Layer Security (TLS,

described in Chapter 9) do not provide all of the necessary requirements. XML security is

intended to bridge this gap.

Web services are increasingly important in distributed systems: they support

interoperability across the global Internet, including the key area of business-to-business

integration and also the emergent ‘mashup’ culture enabling third-party developers to

creative innovative software on top of the existing service base. Web services also provide

the underlying middleware for both the Grid and cloud computing.398 CHAPTER 9 WEB SERVICES

9.1 Introduction

The growth of the Web in the last two decades (see Figure 1.6) proves the effectiveness

of using simple protocols over the Internet as the basis for a large number of wide-area

services and applications. In particular, the HTTP request-reply protocol (Section 5.2),

allows general-purpose clients, called browsers, to view web pages and other resources

with reference to their Uniform Resource Locators (URLs – see the box below for a note

on URIs, URLs and URNs).

However, the use of a general-purpose browser as client, even with the

enhancements provided by downloaded application-specific applets, restricts the

potential scope of applications. In the original client-server model, both client and server

were functionally specialized. Web services return to this model, in which an

application-specific client interacts with a service with a functionally specialized

interface over the Internet.

Thus, web services provide an infrastructure for maintaining a richer and more

structured form of interoperability between clients and servers. They provide a basis

whereby a client program in one organization may interact with a server in another

organization without human supervision. In particular, web services allow complex

applications to be developed by providing services that integrate several other services.

Due to the generality of their interactions, web services cannot be accessed directly by

browsers.

The provision of web services as an addition to web servers is based on the ability

to use an HTTP request to cause the execution of a program. Recall that when a URL in

an HTTP request refers to an executable program, for example, a search, the result is

produced by that program and returned. In a similar way, web services are an extension

of the Web and can be provided by web servers. However, their servers need not be web

servers. The terms ‘web server’ and ‘web services’ should not be confused: a web server

provides a basic HTTP service, whereas a web service provides a service based on the

operations defined in its interface.

External data representation and marshalling of messages exchanged between

clients and web services is done in XML, which is described in Section 4.3.3. To recap,

XML is a textual representation that, although more bulky than alternative

representations, has been adopted for its readability and the consequent ease of

debugging.

The SOAP protocol (Section 9.2.1) specifies the rules for using XML to package

messages, for example to support a request-reply protocol. Figure 9.1 summarizes the

main points about the communication architecture in which web services operate: a web

service is identified by a URI and can be accessed by clients using messages formatted

URI, URL and URN • The Uniform Resource Identifier (URI) is a general resource

identifier, whose value may be either a URL or a URN. URLs, which include

resource location information such as the domain name of the server of a resource

being named, are well known to all web users. Uniform Resource Names (URNs) are

location-independent – they rely on a lookup service to map them onto the URLs of

resources. URNs are discussed in more detail in Section 13.1.SECTION 9.1 INTRODUCTION 399

in XML. SOAP is used to encapsulate these messages and transmit them over HTTP or

another protocol, for example, TCP or SMTP. A web service deploys service

descriptions to specify the interface and other aspects of the service for the benefit of

potential clients.

The top layer of the figure illustrates the following:

• Web services and applications may be built on top of other web services.

• Some particular web services provide general functionality required for the

operation of a large number of other web services. They include directory

services, security and choreography, all of which are discussed later in this

chapter.

A web service generally provides a service description, which includes an interface

definition and other information, such as the server’s URL. This is used as the basis for

a common understanding between client and server as to the service on offer. Section

9.3 presents the Web Services Description Language (WSDL).

Another common need in middleware is for a naming or directory service to allow

clients to find out about services. Clients of web services have similar needs, but

frequently manage without directory services. For example, they often find out about

services from information on a web page (say, as the result of a Google search).

However, some work has been done to provide a directory service that is suitable for use

within organizations. This is discussed in Section 9.4.

XML security is introduced in Section 9.5. In this approach to security, documents

or parts of documents may be signed or encrypted. A document that has signed or

encrypted elements may then be transmitted or stored; later additions may be made and

these too may be signed or encrypted.

Web services provide access to resources for remote clients, but they do not

provide a means for coordinating their operations with one another. Section 9.7.3

discusses choreography of web services, which is intended to allow one web service to

use predefined patterns of access in using a set of other web services.

The last section of this chapter considers applications of web services, including

support for service-oriented architecture, the Grid and cloud computing.

Figure 9.1 Web services infrastructure and components

Security

Service descriptions (in WSDL)

Applications

Directory service

Web Services

XML

Choreography

SOAP

URIs (URLs or URNs) HTTP, SMTP or other transport400 CHAPTER 9 WEB SERVICES

9.2 Web services

A web service interface generally consists of a collection of operations that can be used

by a client over the Internet. The operations in a web service may be provided by a

variety of different resources, for example, programs, objects or databases. A web

service may be managed by a web server along with web pages; or it may be a totally

separate service.

The key characteristic of most web services is that they can process XMLformatted SOAP messages (see Section 9.2.1). An alternative is the REST approach

which is outlined in the box on page 402. Each web service uses its own service

description to deal with the service-specific characteristics of the messages it receives.

For a good account of many more-detailed aspects of web services, see Newcomer

[2002] or Alonso et al. [2004].

Many well-known commercial web servers including Amazon, Yahoo, Google

and eBay, offer web service interfaces that allow clients to manipulate their web

resources. As an example, the web service offered by Amazon.com provides operations

to allow clients to get information about products, to add an item to a shopping cart or

to check the status of a transaction. The Amazon web services [associates.amazon.com]

may be accessed either by SOAP or by REST. This enables third-party applications to

build value-added services over those provided by Amazon.com. For example, an

inventory control and purchasing application might order supplies of various

commodities as they are needed from Amazon.com and automatically keep track of the

changing status of each order. Over 50,000 developers registered to use these web

services in the first two years after they were introduced [Greenfield and Dornan 2004].

Another interesting example of an application that requires the presence of a web

service is one that implements ‘sniping’ in eBay auctions – that is, placing a bid during

the last few seconds before an auction closes. Although humans can perform the same

actions by direct interaction with the web page, they cannot do it as quickly.

Combination of web services • Providing an interface for a web service allows its

operations to be combined with those of other services to provide new functionality (see

also Section 9.7.1). The purchasing application mentioned above might be using other

suppliers as well. As another example of the benefits of combining several services,

consider the fact that many people book flights, hotels and rental cars for trips online

using a variety of different web sites. If each of these web sites were to provide a

standard web service interface, then a ‘travel agent service’ could use their operations

to provide a traveller with a combination of these services. This point is illustrated in

Figure 9.2.

Communication patterns • The travel agent service illustrates the possible use of the two

alternative communication patterns available in web services:

• The processing of a booking takes a long time to complete and could well be

supported by an asynchronous exchange of documents, starting with the details of

the dates and destinations, followed by a return of status information from time to

time and eventually the details of completion. Performance is not an issue here.SECTION 9.2 WEB SERVICES 401

• The checking of credit card details and the interactions with the client should be

supported by a request-reply protocol.

In general, web services either use a synchronous request-reply pattern of

communication with their clients or communicate by means of asynchronous messages.

The latter style of communication may be used even when requests require replies, in

which case the client sends a request and then later receives the reply asynchronously.

An event-style pattern can also be used: for example, clients of a directory service may

register for events of interest and will be notified whenever certain events (such as the

arrival or departure of a service) occur.

To allow for a variety of patterns of communication, the SOAP protocol

(discussed in Section 9.2.1) is based on the packaging of single one-way messages. Its

supports request-reply interactions by using pairs of single messages and specifying

how to represent operations, their arguments and their results.

More generally, web services are designed to support distributed computing in the

Internet, in which many different programming languages and paradigms coexist.

Hence, they are designed to be independent of any particular programming paradigm.

This is in contrast to, for example, distributed objects which advocate a rather specific

programming paradigm for developers (further discussion of the distinctions between

web services and distributed objects can be found in Section 9.2.2).

Loose coupling • There is considerable interest in loose coupling in distributed systems,

particularly in the web services community. The terminology is often ill-defined and

imprecise, though. In the context of web services, loose coupling refers to minimizing

the dependencies between services in order to have a flexible underlying architecture

(reducing the risk that a change in one service will have a knock-on effect on other

services). This is partially supported by the intended independence of web services with

the subsequent intention to produce combinations of web services as discussed above.

Loose coupling is, however, further enhanced by a number of additional features:

• Programming with interfaces (as discussed in Chapter 5) provides one level of

loose coupling by separating the interface from its implementation (and also

supports important areas of heterogeneity, – for example in the choice of

programming language and platform used). Programming with interfaces is

hotel bookinga

Figure 9.2 The ‘travel agent service’ combines other web services

flight bookinga

hire car bookinga

flight bookingb

hotel bookingb

hire car bookingb

Client Travel Agent

Service402 CHAPTER 9 WEB SERVICES

adopted by most distributed systems paradigms including distributed objects and

components (discussed in Chapter 8) as well as web services.

• There is a trend towards simple, generic interfaces in distributed systems and this

is exemplified by the minimal interface offered by the World Wide Web and the

REST approach in web services. This approach contributes to loose coupling by

reducing dependency on specific operation names (the Google case study in

Chapter 21 provides a further example of this style of distributed programming).

One consequence of this is that data becomes more important than operation, with

the semantics of interoperation often held in the data (for example, the associated

XML document definition in web services); this data-oriented view is discussed

further in the context of mobile systems Section 19.3.2.

• As mentioned above, web services can be used with a variety of communication

paradigms, including request-reply communication, asynchronous messaging or

indeed indirect communication paradigms (as featured in Chapter 6). The level of

coupling is directly affected by this choice. For example, in request-reply

communication, the two parties are intrinsically coupled; asynchronous

messaging offers a degree of decoupling (referred to as synchronization

uncoupling in Chapter 6), whereas indirect communication also offers time and

space uncoupling.

In conclusion, there are a number of dimensions to loose coupling, and it is important to

bear this in mind when using the term. Web services intrinsically support a level of loose

coupling due to the design philosophy adopted and the programming with interfaces

approach used. This can be further enhanced by additional design choices, including the

adoption of the REST approach and the use of indirect communication.

Representation of messages • Both SOAP and the data it carries are represented in

XML, a textual self-describing format introduced in Section 4.3.3. Textual

representations take up more space than binary ones and the parsing that they require

takes more time to process. In document-style interactions speed is not an issue, but it is

important in request-reply interactions. However, it is argued that there is an advantage

in a human-readable format that allows for the easy construction of simple messages and

for debugging of more complex ones.

Each item in an XML description is annotated with its type and the meaning of

each type is defined by a schema referenced within the description. This makes the

REST (Representational State Transfer) • REST [Fielding 2000] is an approach with

a very constrained style of operation, in which clients use URLs and the HTTP

operations GET, PUT, DELETE and POST to manipulate resources that are

represented in XML. The emphasis is on the manipulation of data resources rather

than on interfaces. When a new resource is created, it has a new URL by which it can

be accessed or updated. Clients are supplied with the entire state of a resource instead

of calling an operation to get some part of it. Fielding argues that in the context of the

Internet, the proliferation of different service interfaces will not be as useful as a

simple minimum uniform set of operations. It is interesting to note that, according to

Greenfield and Dornan [2004], 80% of the requests to the web services at

Amazon.com are via the REST interface, with the remaining 20% using SOAP.SECTION 9.2 WEB SERVICES 403

format extensible, enabling any type of data to be transported. There is no limit to the

potential richness and complexity of documents formatted in XML, but there could be a

problem in interpreting those that become unduly complex.

Service references • In general, each web service has a URI, which clients use to refer

to it. The URL is the most frequently used form of URI. Because a URL contains the

domain name of a computer, the service to which it refers will always be accessed at that

computer. However, the access point of a web service with a URN can depend on

context and can change from time to time – its current URL can be obtained from a URN

lookup service. This service reference is known as an endpoint in web services.

Activation of services • A web service will be accessed via the computer whose domain

name is included in its current URL. That computer may run the web service itself or it

may run it on another server computer. For example, a service with tens of thousands of

clients may need to be deployed on hundreds of computers. A web service may run

continuously, or it may be activated on demand. The URL is a persistent reference,

meaning that it will continue to refer to the service for as long as the server the URL

points to exists.

Transparency • A major task of many middleware platforms is to protect the

programmer from the details of data representation and marshalling; another is to make

remote invocations look like local ones. None of these things are provided as a part of

an infrastructure or middleware platform for web services. At the simplest level, clients

and servers may read and write their messages directly in SOAP, using XML.

But for convenience, the details of SOAP and XML are generally hidden by a

local API in a programming language such as Java, Perl, Python or C++. In this case,

the service description may be used as a basis for automatically generating the necessary

marshalling and unmarshalling procedures.

Proxies: One way to hide the difference between local and remote calls is by providing

a client proxy or a set of stub procedures. Section 9.2.3 explains how this is done in Java.

Client proxies or stubs provide a static form of invocation in which the framework for

each call and the marshalling procedures are generated before any invocations are made.

Dynamic invocation: An alternative to proxies is to provide clients with a generic

operation to be used irrespective of the remote procedure to be called, similar to the

DoOperation procedure defined in Figure 5.3 (but without the first argument). In this

case, the client specifies the name of an operation and its arguments and they are

converted to SOAP and XML on the fly. The asynchronous communication of single

messages can be achieved in a similar way by providing clients with generic operations

for sending and receiving messages.

9.2.1 SOAP

SOAP is designed to enable both client-server and asynchronous interaction over the

Internet. It defines a scheme for using XML to represent the contents of request and

reply messages (see Figure 5.4) as well as a scheme for the communication of

documents. Originally SOAP was based only on HTTP, but the current version is

designed to use a variety of transport protocols including SMTP, TCP or UDP. The404 CHAPTER 9 WEB SERVICES

description in this section is based on SOAP version 1.2 [www.w3.org IX], which is a

World Wide Web Consortium (W3C) recommendation. SOAP is an extension of

Userland’s XML-RPC [Winer 1999].

The SOAP specification states:

• how XML is to be used to represent the contents of individual messages;

• how a pair of single messages can be combined to produce a request-reply pattern;

• the rules as to how the recipients of messages should process the XML elements

that they contain;

• how HTTP and SMTP should be used to communicate SOAP messages. It is

expected that future versions of the specification will define how to use other

transport protocols, for example, TCP.

This section describes how SOAP uses XML to represent messages and HTTP to

communicate them. However, the programmer does not normally need to be concerned

with these details, since SOAP APIs have been implemented in many programming

languages, including Java, JavaScript, Perl, Python, .NET, C, C++, C# and Visual Basic.

To support client-server communication, SOAP specifies how to use the HTTP

POST method for the request message and its response for the reply message. The

combined use of XML and HTTP provides a standard protocol for client-server

communication over the Internet.

It is intended that a SOAP message can be passed via intermediaries on the way

to the computer that manages the resource to be accessed and that higher-level

middleware services such as transactions or security may use these intermediaries to

perform processing.

SOAP messages • A SOAP message is carried in an ‘envelope’. Inside the envelope

there is an optional header and a body, as shown in Figure 9.3. Message headers can be

used for establishing the necessary context for a service or for keeping a log or audit of

operations. An intermediary may interpret and act on the information in the message

Figure 9.3 SOAP message in an envelope

envelope

header

body

header element

body element

header element

body elementSECTION 9.2 WEB SERVICES 405

headers, for example by adding, altering or removing information. The message body

carries an XML document for a particular web service.

The XML elements envelope, header and body, together with other attributes and

elements of SOAP messages, are defined as a schema in the SOAP XML namespace.

The definition of this schema can be found on the W3C web site [www.w3.org IX].

Since they use a textual encoding, XML schemas can be viewed with the ‘view source’

option of a browser. Both the header and the body contain inner elements.

The previous section explained that service descriptions contain information that

is to be shared by clients and servers. Message senders use these descriptions to generate

the body and to ensure that it contains the correct contents, and message recipients use

them to parse and check the validity of the contents.

A SOAP message may be used either to convey a document or to support clientserver communication:

• A document to be communicated is placed directly inside the body element

together with a reference to an XML schema containing the service description –

which defines the names and types used in the document. This sort of SOAP

message may be sent either synchronously or asynchronously.

• For client-server communication, the body element contains either a Request or a

Reply. These two cases are illustrated in Figure 9.4 and Figure 9.5.

Figure 9.4 shows an example of a simple request message without a header. The body

encloses an element containing the name of the procedure to be called and the URI of

the namespace (the file containing the XML schema) for the relevant service

description, which is denoted by m. The inner elements of a request message contain the

arguments of the procedure. This request message provides two strings to be returned in

the opposite order by the procedure at the server. The XML namespace denoted by env

contains the SOAP definitions for an envelope. Figure 9.5 shows the corresponding

successful reply message, which contains the two output arguments. Note that the name

of the procedure has ‘Response’ added to it. If a procedure has a return value, then it may

be denoted as an element called rpc:result. The reply message uses the same two XML

Figure 9.4 Example of a simple request without headers

m:exchange

env:envelope xmlns:env =namespace URI for SOAP envelopes

m:arg1

env:body

xmlns:m = namespace URI of the service description

Hello

In this figure and the next, each XML element is represented by a shaded box with its

m:arg2

World

name in italics,at the top left corner,followed by any attributes and its content406 CHAPTER 9 WEB SERVICES

schemas as the request message, the first defining the SOAP envelope and the second

the application-specific procedure and argument names.

Soap faults: If a request fails in some way, the fault descriptions are conveyed in the

body of a reply message in a fault element. This element contains information about the

fault, including a code and an associated string, together with application-specific

details.

SOAP headers • Message headers are intended to be used by intermediaries to add to

the service that deals with the message carried in the corresponding body. However, two

aspects of this usage are left unclear in the SOAP specification:

1. How the headers will be used by any particular higher middleware service. For

example, a header might contain:

– a transaction identifier for use with a transaction service;

– a message identifier for relating messages to one another, for example, for

implementing reliable delivery;

– a username, a digital signature or a public key.

2. How the messages will be routed via a set of intermediaries to the ultimate

recipient. For example, a message transported by HTTP could be routed via a

chain of proxy servers, some of which might assume a SOAP role.

However, the specification does specify the roles and duties of intermediaries. An

attribute called role can specify whether every intermediary, none of them, or just the

ultimate recipient must process the element [www.w3.org IX]. The particular actions to

be carried out are defined by applications – for example, an action might be to log the

contents of an element.

Transport of SOAP messages • A transport protocol is required to send a SOAP

message to its destination. SOAP messages are independent of the type of transport used

– their envelopes contain no reference to the destination address. HTTP (or whatever

protocol is used to transport a SOAP message) is left to specify the destination address.

Figure 9.6 illustrates how the HTTP POST method is used to transmit a SOAP

message. The HTTP headers and body are used as follows:

Figure 9.5 Example of a reply corresponding to the request in Figure 9.4

env:envelope xmlns:env = namespace URI for SOAP envelope

m:res1

env:body

xmlns:m = namespace URI for the service description

m:res2

World

m:exchangeResponse

HelloSECTION 9.2 WEB SERVICES 407

• The HTTP headers specify the endpoint address (the URI of the ultimate receiver)

and the action to be carried out. The Action header is intended to optimize

dispatching by revealing the name of the operation without the need to analyze the

SOAP message in the body of the HTTP message.

• The HTTP body carries the SOAP message.

As HTTP is a synchronous protocol, it is used to return a reply containing the SOAP

reply, like the one shown in Figure 9.5. Section 5.2 details the status codes and reasons

returned by HTTP for successful and failing requests.

If a SOAP Request is just a request for information to be returned, has no

arguments and does not alter data in the server, then the HTTP GET method can be used

to carry it out.

The above point about the Action header and dispatching applies to any service

that performs a variety of different actions for clients, even if it does not offer operations

as such. For example, a web service may be able to deal with various types of

documents, such as purchase orders and enquiries, which are dealt with by different

software modules. The Action header enables the correct module to be chosen without

inspecting the SOAP message. This header can be used if the HTTP content type is

specified as application/soap+xml.

The separation of the definition of the SOAP envelope from the information as to

how and where it is to be sent makes it possible to use a variety of different underlying

protocols. For example, the SOAP specification states how SMTP can be used as an

alternative way of transmitting documents encoded as SOAP messages.

But this strength is also a weakness. It implies that the developer must be involved

in the details of the specific transport protocol chosen. In addition, it makes it difficult

to use different protocols for different parts of the route followed by a particular

message.

WS-Addressing: Advances in SOAP addressing and routing • Two problems were mentioned above:

• how to make SOAP independent of the underlying transport used;

• how to specify a route to be followed by a SOAP message via a set of

intermediaries.

Figure 9.6 Use of HTTP POST Request in SOAP client-server communication

endpoint address

action

POST /examples/stringer

Host: www.cdk4.net

Content-Type: application/soap+xml

Action: http://www.cdk4.net/examples/stringer#exchange

<env:envelope xmlns:env = namespace URI for SOAP envelope>

<env:header> </env:header>

<env:body> </env:body>

</env:Envelope>

SOAP message HTTP headers408 CHAPTER 9 WEB SERVICES

Early work in this area by Nielsen and Thatte [2001] suggests that the endpoint address

and the dispatching information should be specified in SOAP headers. This effectively

separates the message destination from the underlying protocol. They suggested

specifying the path to be followed by giving the address of the endpoint and the ‘next

hop’. Each of the intermediaries would update the ‘next hop’ information.

The work of Box and Curbera [2004] suggests that having intermediaries alter the

headers could lead to breaches of security. They proposed WS-Addressing, which

allows SOAP headers to specify message routing data, with an underlying SOAP

infrastructure providing the ‘next hop’ information. The W3C recommendations for

WS-Addressing are defined in [www.w3.org XXIII]. This form of addressing uses an

Endpoint Reference – an XML structure containing the destination address, routing

information and possibly other information about the service. To support long-running

asynchronous interactions, SOAP headers can supply a return address and message

identifiers of their own and of related messages.

WS-ReliableMessaging: Reliable communication • SOAP’s usual protocol, HTTP,

runs over TCP, whose failure model is discussed in Section 4.2.4. To summarize: TCP

does not guarantee to deliver messages in the face of all difficulties, and when it times

out while waiting for acknowledgements, it declares that the connection is broken, at

which point the communicating processes are left without any idea as to whether the

messages they sent recently have been received or not.

Early work on the provision of reliable communication of SOAP messages with

guaranteed delivery, no duplicates and guaranteed message ordering led to two

competing specifications by Ferris and Langworthy [2004] and Evans et al. [2003].

More recently, Oasis (a global consortium that works on the development,

agreement and adoption of e-business and web service standards) has made a

recommendation called WS-ReliableMessaging [www.oasis.org]. This allows a SOAP

message to be delivered at-least-once, at-most-once or exactly-once, with the following

semantics:

At-least-once: The message is delivered at least once, but an error is reported if it

cannot be delivered.

At-most-once: The message is delivered at most once, but without any error report if

it cannot be delivered.

Exactly-once: The message is delivered exactly once, but an error is reported if it

cannot be delivered.

Ordering of messages is also provided in combination with any of the above:

In-order: Messages will be delivered to the destination in the order in which they

were sent by a particular sender.

Note that WS-ReliableMessaging is concerned with the delivery of single messages and

should not be confused with the RPC call semantics described in Section 5.3.1, which

refer to the number of times the server executes the remote procedure. The reader is

referred to Exercise 9.16 for further consideration of the comparison.

Traversing firewalls • Web services are intended to be used by clients in one

organization to access servers in another organization over the Internet. MostSECTION 9.2 WEB SERVICES 409

organizations use a firewall to protect the resources on their own networks, and

transport protocols such as those used by Java RMI or CORBA will not normally be able

to pass through a firewall. However, firewalls do normally allow both HTTP and SMTP

messages to pass through them. Therefore it is convenient to use one of these protocols

for transporting SOAP messages.

9.2.2 A comparison of web services with the distributed object model

A web service has a service interface that can provide operations for accessing and

updating the data resources it manages. At a superficial level, the interaction between

client and server is very similar to RMI, where a client uses a remote object reference to

invoke an operation in a remote object. For a web service, the client uses a URI to invoke

an operation in the resource named by that URI. For arguments about the similarities and

differences between web services and distributed objects, see Birman [2004], Vinoski

[2002] and Vogels [2003].

We shall attempt to show that there are limits to the above analogy, by making use

of the shared whiteboard example used in Section 5.5 for Java RMI and Section 8.3 for

CORBA.

Remote object references versus URIs • The URI of a web service can be compared

with the remote object reference of a single object. However, in the distributed object

model, objects can create remote objects dynamically and return remote references to

them. The recipient of these remote references can use them to invoke operations in the

objects to which they refer. In the shared whiteboard example, an invocation of the

newShape factory method causes a new instance of Shape to be created and a remote

reference to it is returned. Nothing like this can be done with web services, which cannot

create instances of remote objects; effectively, a web service consists of a single remote

object and therefore both garbage collection and remote object referencing are

irrelevant.

Web services model • The users of the Java web services toolkit (JAX-RPC)

[java.sun.com VII] must model their web services programs to allow for the fact that

they are not using transparent Java-to-Java remote invocation but rather are using the

web services model, in which remote objects cannot be instantiated. This is taken into

account by JAX-RPC, which does not permit remote object references to be passed as

arguments or returned as results.

Figure 9.7 shows a version of the interface given in Figure 5.16 which has been

modified as follows to become a web services interface:

• In the original (distributed object) version of the program, instances of Shape are

created in the server and remote references to them are returned by newShape,

whose modified (web service) version is shown in line 1. To avoid the

instantiation of remote objects and the consequent use of remote object references,

the Shape interface is removed and its operations (getAllState and getGOVersion

– originally getVersion) are added to the ShapeList interface.

• In the original (distributed object) version of the program, the server stored a

vector of Shape. This will be changed to a vector of GraphicalObject. The new410 CHAPTER 9 WEB SERVICES

(web service) version of the method newShape returns an integer that gives the

offset of the GraphicalObject in that vector.

This change to the method newShape means that it is no longer a factory method – that

is, it does not create instances of remote objects.

Servants • In the distributed object model, the server program is generally modelled as

a collection of servants (potentially remote objects). For example, the shared whiteboard

application used one servant for the list of shapes and one servant for each graphical

object created. These servants were created as instances of the servant classes ShapeList

and Shape, respectively. When the server started, its main function created the instance

of ShapeList, and each time the client called the newShape method the server created an

instance of Shape.

In contrast, web services do not support servants. Therefore web services

applications cannot create servants as and when they are needed to handle different

server resources. To enforce this situation, the implementations of web service

interfaces must not have either constructors or main methods.

9.2.3 The use of SOAP with Java

The Java API for developing web services and clients over SOAP is called JAX-RPC.

It is described in the Java web services tutorial [java.sun.com VII]. This API hides all

the details of SOAP from the programmers of both clients and the services.

JAX-RPC maps some of the types in the Java language to definitions in XML used

in both SOAP messages and service descriptions. The permitted types include Integer,

String, Date and Calendar, as well as java.net.uri, which allows URIs to be passed as

arguments or returned as results. It supports some of the collection types (including

Vector) as well as the primitive types of the language and arrays.

In addition, instances of some classes may be passed as arguments and results of

remote calls, provided that:

• Each of their instance variables is one of the permitted types.

• They have a public default constructor.

• They do not implement the Remote interface.

Figure 9.7 Java web service interface ShapeList

import java.rmi.\*;

public interface ShapeList extends Remote {

int newShape(GraphicalObject g) throws RemoteException; 1

int numberOfShapes() throws RemoteException;

int getVersion() throws RemoteException;

int getGOVersion(int i) throws RemoteException;

GraphicalObject getAllState(int i) throws RemoteException;

}SECTION 9.2 WEB SERVICES 411

In general, as mentioned in the previous section, values of types that are remote

references (that is, that implement the Remote interface) cannot be passed as arguments

or returned as results of remote calls.

The service interface • The Java interface of a web service must conform to the

following rules, some of which are illustrated in Figure 9.7:

• It must extend the Remote interface.

• It must not have constant declarations, such as public final static.

• The methods must throw java.rmi.RemoteException or one of its subclasses.

• Method parameters and return types must be permitted JAX-RPC types.

The server program • The class that implements the interface ShapeList is shown in

Figure 9.8. As explained above, there is no main method, and the implementation of the

ShapeList interface does not have a constructor. In effect, a web service is a single object

that offers a set of procedures. The source of the programs shown in Figure 9.7, Figure

9.8 and Figure 9.9 is available on the book’s web site at www.cdk5.net/web.

The service interface and its implementation are compiled as usual. A pair of tools

called wscompile and wsdeploy can be used to generate the skeleton class and the service

description (in WSDL, as described in Section 9.3), using information concerning the

URL of the service, its name and description retrieved from a configuration file written

in XML. The name of the service (in this case, MyShapeListService) is used to generate

the name of the class used in the client program to access it – that is,

MyShapeListService\_Impl.

Figure 9.8 Java implementation of the ShapeList server

import java.util.Vector;

public class ShapeListImpl implements ShapeList{

private Vector theList = new Vector();

private int version = 0;

private Vector theVersions = new Vector();

public int newShape(GraphicalObject g) throws RemoteException{

version++;

theList.addElement(g);

theVersions.addElement(new Integer(version));

return theList.size();

}

public int numberOfShapes(){}

public int getVersion() {}

public int getGOVersion(int i){}

public GraphicalObject getAllState(int i) {}

}412 CHAPTER 9 WEB SERVICES

Servlet container • The service implementation is run as a servlet inside a servlet

container whose role is to load, initialize and execute servlets. The servlet container

includes a dispatcher and skeletons (see Section 5.4.2). When a request arrives, the

dispatcher maps it to a particular skeleton, which translates it into Java and passes on the

request to the appropriate method in the servlet. That method carries out the request and

produces a reply, which the skeleton translates back into a SOAP reply. The URL of a

service consists of a concatenation of the URL of the servlet container and the service

category and name, for example, http://localhost:8080/ShapeList-jaxrpc/ShapeList.

Tomcat [jakarta.apache.org] is a commonly used servlet container. When Tomcat

is running, its management interface is available at a URL for viewing with a browser.

This interface shows the names of servlets that are currently deployed and provides

operations for managing them and for accessing information about each one, including

its service description. Once a servlet is deployed in Tomcat, clients may access it and

the combined effects of their operations will be stored in its instance variables. In our

example, a list of GraphicalObjects will be built up as each one is added as the result of

a client request to the newShape operation. If a servlet is stopped by Tomcat’s

management interface, then the values of the instance variables are reset when it is

restarted.

Tomcat also provides access to a description of each of the services that it

contains, to enable programmers to design client programs and to facilitate the

automatic compilation of the proxies required by client code. The service description is

human-readable since it is expressed in XML notation (more specifically, in WSDL, as

introduced in Section 9.3).

Note that it is possible to develop web services without using servlet containers;

for example, Apache Axis hides this level of detail from the programmer.

The client program • The client program may use static proxies, dynamic proxies or a

dynamic invocation interface. In all cases, the relevant service description may be used

to obtain any information required by client code. In our example, the service

description can be obtained from Tomcat.

Static proxies: Figure 9.9 shows the ShapeList client making a call through a proxy – a

local object that passes on messages to the remote service. The code for the proxy is

generated by wscompile from the service description. The class name for the proxy is

formed by adding ‘\_Impl’ to the name of the service – in this case, the proxy class is

called MyShapeListService\_Impl. This name is implementation-specific, since the

SOAP specification does not give a rule for naming proxy classes.

At line 1, the createProxy method is called. This method is shown at line 5; line 6

creates a proxy, using the class MyShapeListService\_Impl, which it returns (note that

proxies are sometimes called stubs, hence the name of the class, Stub). At line 2, the

URL of the service is supplied to the proxy via the argument given on the command line.

At line 3, the type of the proxy is narrowed to suit the type of the interface – ShapeList.

Line 4 makes a call to the remote procedure getAllState, asking the service to return the

object at element 0 in the vector of GraphicalObjects.

Because the proxy is created at compile time, it is called a static proxy. The service

description of the service from which it was generated will not necessarily have been

generated from a Java interface, but may have been made by any one of a variety of toolsSECTION 9.2 WEB SERVICES 413

associated with various different language systems. It may even have been written

directly in XML.

Dynamic proxies: Instead of using a precompiled static proxy, the client may use a

dynamic proxy whose class is created at runtime from the information in the service

description and the interface of the service. This method avoids the need for involving

an implementation-specific name for the proxy class.

Dynamic invocation interface: This allows a client to call a remote procedure, even if its

signature or the name of the service is unknown until runtime. In contrast to the above

alternatives, the client does not require a proxy. Instead, it has to use a series of

operations to set the name of the server operation, the return value and each of the

parameters before making the procedure call.

Implementation of Java SOAP • The way the Java API is implemented can be explained

with reference to Figure 5.15. The following paragraphs explain the roles of the various

components in a Java/SOAP environment – the interactions between the components are

the same as before. There is no remote reference module.

Communication modules: The tasks of these modules are carried out by a pair of HTTP

modules. The HTTP module in the server selects the dispatcher according to the URL

given in the Action header to the POST request.

Client proxy: A proxy (or stub) method knows the URL of the service and marshals its

own method name and its arguments, together with a reference to the XML schema for

the service, into a SOAP request envelope. Unmarshalling the reply consists of

analyzing a SOAP envelope in order to extract the results, return value or fault report.

The client’s request method call is sent to the service as an HTTP request.

Figure 9.9 Java implementation of the ShapeList client

package staticstub;

import javax.xml.rpc.Stub;

public class ShapeListClient {

public static void main(String[] args) { /\* pass URL of service \*/

try {

Stub proxy = createProxy(); 1

proxy.\_setProperty 2

(javax.xml.rpc.Stub.ENDPOINT\_ADDRESS\_PROPERTY, args[0]);

ShapeList aShapeList = (ShapeList)proxy; 3

GraphicalObject g = aShapeList.getAllState(0); 4

} catch (Exception ex) { ex.printStackTrace(); }

}

private static Stub createProxy() { 5

return

(Stub) (new MyShapeListService\_Impl().getShapeListPort()); 6

}

}414 CHAPTER 9 WEB SERVICES

Dispatcher and skeleton: As mentioned above, the dispatcher and skeletons live in the

servlet container. The dispatcher extracts the name of the operation from the Action

header in the HTTP request and invokes the corresponding method in the appropriate

skeleton, passing the SOAP envelope to it. A skeleton method carries out the following

tasks: it analyzes the SOAP envelope in the request message and extracts its arguments,

calls the corresponding method and assembles a SOAP reply envelope containing the

results.

Errors, faults and correctness in SOAP/XML: Faults may be reported by the HTTP

module, the dispatcher, the skeleton or the service itself. The service can report its errors

via a return value or by means of fault parameters specified in the service description.

The skeleton is responsible for checking that the SOAP envelope contains a request and

that the XML in which it is written is well-formed. Having established that the XML is

well-formed, the skeleton will use the XML namespace in the envelope to check that the

request corresponds to the service on offer and that the operation and its arguments are

appropriate. If the checking of the request fails at either of these levels, then an error is

returned to the client. Similar checks are made by the proxy when it receives the SOAP

envelope containing the result.

9.2.4 Comparison of web services with CORBA

The main difference between web services and CORBA or other similar middleware is

the intended usage context. CORBA was designed for use within a single organization

or between a small number of collaborating organizations. This resulted in certain

aspects of the design being too centralized for collaborative use by independent

organizations or for ad hoc use without prior arrangements, as will now be explained.

Naming issues • In CORBA, each remote object is referenced by means of a name that

is managed by an instance of the CORBA Naming Service (see Section 8.3.5). This

service, like the DNS, provides a mapping from a name to a value to be used as an

address (an IOR in CORBA). But unlike the DNS, the CORBA Naming Service is

designed for use within an organization, instead of throughout the Internet.

In the CORBA Naming Service, each server manages a graph of names with an

initial naming context and is initially independent of any other servers. Although

separate organizations may federate their naming services, this is not automatic. Before

a server can federate with another, it needs to know the other server’s initial naming

context. Thus, the design of the CORBA Naming Service effectively restricts the

sharing of CORBA objects to within a small set of organizations that have federated

their naming services.

Reference issues • We now consider whether a CORBA remote object reference,

which is called an IOR (Section 8.3.3), could be used as an Internet-wide object

reference in the same way as a URL. Each IOR contains a slot that specifies the type

identifier of the interface of the object it references. However, this type identifier is

understood only by the interface repository that stores the definition of the

corresponding type. This has the implication that the client and server need to use the

same interface repository, which is not really practical on a global scale.SECTION 9.2 WEB SERVICES 415

In contrast, in the web services model, a service is identified by means of a URL,

enabling a client anywhere in the Internet to make a request to a service that may belong

to any organization anywhere else. That is, a web service can be shared by clients

throughout the Internet. The DNS is the only service required for URL access, and it is

designed to work effectively Internet-wide.

Separation of activation and location • The tasks of locating and activating web services

are neatly separated. In contrast, a CORBA persistent reference refers to a component

of the platform (the implementation repository) that activates the corresponding object

on demand on any suitable computer and is also responsible for locating the object once

it has been activated.

Ease of use • The HTTP and XML infrastructure for web services is well understood

and convenient to use and is already installed on all of the most commonly used

operating systems, although the user does require a convenient programming language

API to SOAP. In contrast, the CORBA platform is a large and complex piece of software

requiring installation and support.

Efficiency • CORBA has been designed to be efficient: CORBA CDR (Section 4.3.1) is

binary, whereas XML is textual. A study by Olson and Ogbuji [2002] compared the

performance of CORBA with that of SOAP and XML-RPC. They found that SOAP

request messages are 14 times as large as the equivalent ones in CORBA and that a

SOAP request took, on average, 882 times as long as an equivalent CORBA invocation.

Although the relative performance is dependent on the language used and also the

particular middleware implementation employed, this example does provide an

indication of the potential overhead of XML-based approaches. For many applications,

however, the message overhead and slower performance of SOAP are not noticed, and

its effects are made less obvious by the availability of cheap bandwidth, processors,

memory and disk space.

The W3C and others have been investigating the possibility of allowing binary

data to be included in XML elements, so as to increase efficiency. Discussions of this

topic can be found at [www.w3.org XXI] and [www.w3.org XXII]. Note that XML does

already provide for both hexadecimal and base64 representations of binary data. The

base64 representation is used in conjunction with XML encryption (see Section 9.5).

There is a considerable time and space overhead when binary data is converted to base64

or hexadecimal, so what is really needed is to be able to include a binary representation

of a preparsed sequence of data items such as, for example, that produced by CORBA

CDR or gzip. Another approach that is also under investigation is to take a SOAP

message – together with attachments, some of which may be binary – and use a multipart

MIME document to transport it.

The strengths of CORBA • The availability of CORBA services for transactions,

concurrency control, security and access control, events and persistent objects makes it

a desirable choice for use in many applications that are intended for use within an

organization or a related group of organizations. In general, it is a good choice for those

applications that require very complex interactions. In addition, the distributed object

model is an attractive one for the design of complex applications, and it is worth the

extra learning effort needed to understand the details of the relationship between the

CORBA object model (Section 8.3) and the particular programming language in use.416 CHAPTER 9 WEB SERVICES

9.3 Service descriptions and IDL for web services

Interface definitions are needed to allow clients to communicate with services. For web

services, interface definitions are provided as part of a more general service description,

which specifies two other additional characteristics – how the messages are to be

communicated (for example, by SOAP over HTTP) and the URI of the service. To cater

for use in a multi-language environment, service descriptions are written in XML.

A service description forms the basis of an agreement between a client and a

server as to the service on offer. It assembles all of the facts concerning the service that

are relevant to its clients. Service descriptions are generally used to generate client stubs

that automatically implement the correct behaviour for the client.

The IDL-like component of a service description is more flexible than other IDLs,

in that a service may be specified either in terms of the types of messages that it will send

and receive or in terms of the operations it supports, to allow for both document

exchange and request-reply-style interactions.

A variety of different methods of communication can be used by web services and

their clients. Therefore the method of communication is left to be decided by the service

provider and specified in the service description, rather than built into the system, as it

is in CORBA, for example.

The ability to specify the URI of a service as a part of the service description

avoids the need for the separate binder or naming service used by most other

middleware. It has the implication that the URI cannot be changed once the service

description has been made available to potential clients, but the URN scheme does cater

for a change of location by allowing for an indirection at the reference level.

In contrast, in the binder approach, the client uses a name to look up the service

reference at runtime, allowing the service references to change over time. This approach

requires an indirection from a name to a service reference for all services, even though

many of them may always remain at the same location.

In the web services context, the Web Services Description Language (WSDL) is

commonly used for service descriptions. The current version, WSDL 2.0 [www.w3.org

XI], became a W3C Recommendation in 2007. It defines an XML schema for

representing the components of a service description, which include, for example, the

element names definitions, types, message, interface, bindings and services.

WSDL separates the abstract part of a service description from the concrete part,

as shown in Figure 9.10.

Figure 9.10 The main elements in a WSDL description

abstract concrete

how where

definitions

types

target namespace

message interface bindings services

document-style request-reply-styleSECTION 9.3 SERVICE DESCRIPTIONS AND IDL FOR WEB SERVICES 417

The abstract part of the description includes a set of definitions of the types used

by the service – in particular, the types of the values exchanged in messages. The Java

example from Section 9.2.3, whose Java interface is shown in Figure 9.7, uses the Java

types int and GraphicalObject. The former (like any basic type) can be translated

directly into the XML equivalent, but GraphicalObject is defined in Java in terms of the

types int, String and boolean. GraphicalObject is represented in XML, for common use

by heterogeneous clients, as a complexType consisting of a sequence of named XML

types including, for example:

<element name="isFilled" type="boolean"/>

<element name="originx" type="int"/>

The set of names defined within the types section of a WSDL definition is called its

target namespace. The message section of the abstract part contains a description of the

set of messages exchanged. For the document style of interaction, these messages will

be used directly. For the request-reply style of interaction, there are two messages for

each operation, which are used to describe the operations in the interface section. The

concrete part specifies how and where the service may be contacted.

The inherent modularity of a WSDL definition allows its components to be

combined in different ways – for example, the same interface may be used with

different bindings or locations. The types may be defined inside the types element or

they may be defined in a separate document referenced by a URI from the types element.

In the latter case, the type definitions can be referenced from several different WSDL

documents.

Messages or operations • In web services, all that the client and the server need is to

have a common idea about the messages to be exchanged. For a service based on the

exchange of a small number of different types of document, WSDL just describes the

types of the different messages to be exchanged. When a client sends one of these

messages to a web service, the latter decides what operation to perform and what type

of message to send back to the client on the basis of the message type received. In our

Java example, two messages will be defined for each of the operations in the interface –

one for the request and one for the reply. For example, Figure 9.11 shows the request

and reply messages for the newShape operation, which has a single input argument of

type GraphicalObject and a single output argument of type int.

For services that support several different operations, it is more effective to

specify the messages exchanged as requests for operations with arguments and their

corresponding replies, allowing the service to dispatch each request to the appropriate

Figure 9.11 WSDL request and reply messages for the newShape operation

message name = "ShapeList\_newShape"

type = "ns:GraphicalObject"

part name="GraphicalObject\_1"

tns – target namespace xsd – XML schema definitions

message name = "ShapeList\_newShapeResponse"

part name="result"

type = "xsd:int"418 CHAPTER 9 WEB SERVICES

operation. However, in WSDL an operation is a construct for relating request and reply

messages, in contrast to the definition of an operation in a service interface.

Interface • The collection of operations belonging to a web service are grouped

together in an XML element named interface (sometimes called portType). Each

operation must specify the message exchange pattern between client and server. The

available options include those shown in Figure 9.12. The first one, In-Out, is the

commonly used request-reply form of client-server communication. In this pattern, the

reply message may be replaced with a fault message. In-Only is for one-way messages

with maybe semantics and Out-Only is for oneway messages from server to client; fault

messages cannot be sent with either. Robust In-Only and Robust Out-Only are the

corresponding messages with guaranteed delivery; fault messages may be exchanged.

Out-In is a request-reply interaction initiated by the server. WSDL 2.0 is also extensible

in that organizations can introduce their own message exchange patterns if the

predefined ones prove to be inadequate.

Returning to our Java example, each of the operations is defined to have an In-Out

pattern. The operation newShape is shown in Figure 9.13, using the messages defined in

Figure 9.11. This definition, together with definitions of the four other operations will

be enclosed in an XML interface element. An operation may also specify the fault

messages that can be sent.

Figure 9.12 Message exchange patterns for WSDL operations

Name Messages sent by

Client Server Delivery Fault message

In-Out Request Reply May replace Reply

In-Only Request No fault message

Robust In-Only Request Guaranteed May be sent

Out-In Reply Request May replace Reply

Out-Only Request No fault message

Robust Out-Only Request Guaranteed May send fault

Figure 9.13 WSDL operation newShape

operation name = "newShape"

input message = "tns:ShapeList\_newShape"

output message = "tns:ShapeList\_newShapeResponse"

pattern = In-Out

tns – target namespace xsd – XML schema definitions

The names operation, pattern, input and output are defined in the XML schema for WSDLSECTION 9.3 SERVICE DESCRIPTIONS AND IDL FOR WEB SERVICES 419

If, for example, an operation has two arguments – say, an integer and a string –

there is no need to define a new data type, since these types are defined for XML

schemas. However, it will be necessary to define a message that has these two parts. This

message can then be used as an input or output in the definition for the operation.

Inheritance: Any WSDL interface may extend one or more other WSDL interfaces. This

is a simple form of inheritance in which an interface supports the operations of any

interfaces it extends in addition to those it defines itself. Recursive definition of

interfaces is not allowed; that is, if interface B extends interface A, then interface A

cannot extend interface B.

Concrete part • The remaining (concrete) part of a WSDL document consists of the

binding (the choice of protocols) and the service (the choice of endpoint or server

address). The two are related, since the form of address depends on the type of protocol

in use. For example, a SOAP endpoint will use a URI whereas a CORBA endpoint will

use a CORBA-specific object identifier.

Binding: The binding section in a WSDL document says which message formats and

form of external data representation are to be used. For example, web services frequently

use SOAP, HTTP and MIME. Bindings may be associated with particular operations or

interfaces, or they may be left free for use by a variety of different web services.

Figure 9.14 shows an example of a binding enclosing a soap:binding that specifies

the URL of a particular protocol for transmitting SOAP envelopes: the HTTP binding

for SOAP. Optional attributes of this element may also specify the following:

• the message exchange pattern, which may be either rpc (request-reply) or

document exchange – the default value is document;

Figure 9.14 SOAP binding and service definitions

soap:binding transport = URI

binding

style= "rpc"

endpoint

service

name =

binding = "tns:ShapeListBinding"

soap:address

location = service URI

name = "MyShapeListService"

name = "ShapeListPort"

for schemas for soap/http

The service URI is:

operation

soap:operation

soapAction

"ShapeListBinding"

type = "tns:ShapeList"

name="newShape"

input

soap:body

encoding, namespace

soap:body

encoding, namespace

output

"http://localhost:8080/ShapeList-jaxrpc/ShapeList"420 CHAPTER 9 WEB SERVICES

• the XML schema for the message formats – the default is the SOAP envelope;

• the XML schema for the external data representation – the default is the SOAP

encoding of XML.

Figure 9.14 also shows the details of the bindings for one of the operations (newShape),

specifying that both the input and the output message should travel in a soap:body, using

a particular encoding style, and that the operation should be transmitted as a soapAction.

Service: Each service element in a WSDL document specifies the name of the service

and one or more endpoints (or ports) where an instance of the service may be contacted.

Each of the endpoint elements refers to the name of the binding in use and, in the case

of a SOAP binding, uses a soap:address element to specify the URI of the service

location.

Documentation • Both human- and machine-readable information may be inserted in a

documentation element at most points within a WSDL document. This information may

be removed before WSDL is used for automatic processing, for example, by stub

compilers.

WSDL use • Complete WSDL documents can be accessed via their URIs by clients and

servers, either directly or indirectly via a directory service such as UDDI. Tools are

available for generating WSDL definitions from information provided via a graphical

user interface, removing the need for users to be involved in the complex details and

structure of WSDL. For example, the Web Services Description Language for Java

Toolkit (WSDL4J) allows the creation, representation and manipulation of WSDL

documents describing services [wsdl4j.sourceforge.org]. WSDL definitions can also be

generated from interface definitions written in other languages, such as Java JAX-RPC,

discussed in Section 9.2.1.

9.4 A directory service for use with web services

There are many ways in which clients can obtain service descriptions. For example,

anyone providing a higher-level web service like the travel agent service discussed in

Section 9.1 would almost certainly make a web page advertising the service and

potential clients would come across the web page when searching for services of that

type.

However, any organization that plans to base its applications on web services will

find it more convenient to use a directory service to make these services available to

clients. This is the purpose of the Universal Description, Discovery and Integration

service (UDDI) [Bellwood et al. 2003], which provides both a name service and a

directory service (see Section 13.3). That is, WSDL service descriptions may be looked

up by name (a white pages service) or by attribute (a yellow pages service). They may

also be accessed directly via their URLs, which is convenient for developers who are

designing client programs that use the service.

Clients may use the yellow pages approach to look up a particular category of

service, such as travel agent or bookseller, or they may use the white pages approach to

look up a service with reference to the organization that provides it.SECTION 9.4 A DIRECTORY SERVICE FOR USE WITH WEB SERVICES 421

Data structures • The data structures supporting UDDI are designed to allow all the

above styles of access and can incorporate any amount of human-readable information.

The data is organized in terms of the four structures shown in Figure 9.15, each of which

can be accessed individually by means of an identifier called a key (apart from tModel,

which can be accessed by a URL):

businessEntity describes the organization that provides these web services, giving its

name, address and activities, etc.;

businessServices stores information about a set of instances of a web service, such as

its name and a description of its purpose (for example, travel agent or bookseller);

bindingTemplate holds the address of a web service instance and references to

service descriptions;

tModel holds service descriptions, usually WSDL documents, stored outside the

database and accessed by means of URLs.

Lookup • UDDI provides an API for looking up services based on two sets of query

operations:

• The get\_xxx set of operations includes get\_BusinessDetail, get\_ServiceDetail,

get\_bindingDetail and get\_tModelDetail; they retrieve an entity corresponding to

a given key.

• The find\_xxx set of operations includes find\_business, find\_service, find\_binding

and find\_tModel; they retrieve the set of entities that matches a particular set of

search criteria, providing a summary of names, descriptions, keys and URLs.

Thus clients in possession of a particular key may use a get\_xxx operation to retrieve the

corresponding entity directly, and other clients may use browsing to assist with searches,

starting with a large set of results and gradually narrowing it down. For example, they

may start by using the find\_business operation in order to get a list containing a summary

tModel

businessServices

tModel

Figure 9.15 The main UDDI data structures

businessEntity

information

about the publisher

tModel

human-readable businessServices

service descriptions

key

URL

URL

URL

businessServices

information

about a

family of services

human-readable

service interfaces

bindingTemplate

bindingTemplate

bindingTemplate

information

about the

key

service interfaces

key422 CHAPTER 9 WEB SERVICES

of information on matching providers. From this summary, the user may use the

find\_service operation to narrow the search by matching the sort of service required. In

both cases, they will find the key of a suitable bindingTemplate and thereby find the

URL for retrieving the WSDL document for a suitable service.

In addition, UDDI provides a notify/subscribe interface by which clients register

interest in a particular set of entities in a UDDI registry and get change notifications,

either synchronously or asynchronously.

Publication • UDDI provides an interface for publishing and updating information

about web services. The first time that a data structure (see Figure 9.15) is published at

a UDDI server, it is given a key in the form of a URI – for example, uddi:cdk5.net:213

– and that server becomes its owner.

Registries • The UDDI service is based on replicated data stored in registries. A UDDI

registry consists of one or more UDDI servers, each of which has a copy of the same set

of data. The data is replicated between the members of a registry. Each of them may

respond to queries and publish information. Changes to a data structure must be

submitted to its owner – that is, the server at which it was first published. It is possible

for an owner to pass on the ownership to another UDDI server in the same registry.

Replication scheme: The members of a registry propagate copies of data structures to one

another as follows: a server that has made changes notifies the other servers in the

registry, which then request the changes. A form of vector timestamp is used to

determine which of the changes should be propagated and applied. The scheme is simple

in comparison with other replication schemes that use vector timestamps, such as Gossip

(Section 18.4.1) or Coda (Section 18.4.3) for two reasons:

1. All changes to a particular data structure are made at the same server.

2. Updates from a particular server are received in sequential order by the other

members, but no particular ordering is imposed between update operations made

by different servers.

Interaction between servers: As described above, servers interact with one another to

carry out the replication scheme. They can also interact in order to transfer ownership of

data structures. However, the response to a lookup operation is made by a single server

without any interaction with other servers in the registry, unlike in the X.500 directory

service (Section 13.5), in which data is partitioned between servers that cooperate with

one another in finding the relevant server for a particular request.

9.5 XML security

XML security consists of a set of related W3C designs for signing, key management and

encryption. It is intended for use in cooperative work over the Internet involving

documents whose contents may need to be authenticated or encrypted. Typically the

documents are created, exchanged, stored and then exchanged again, possibly after

being modified by a series of different users.SECTION 9.5 XML SECURITY 423

WS-Security [Kaler 2002] is another approach to security that is concerned with

applying message integrity, message confidentiality and single message authentication

to SOAP.

As an example of a context in which XML security would be useful, consider a

document containing a patient’s medical records. Different parts of this document are

used at the local doctor’s surgery and at the various special clinics and hospitals visited

by the patient. It will be updated by doctors, nurses and consultants making notes on the

patient’s condition and treatment, by administrators making appointments and by

pharmacists providing medicine. Different parts of the document will be viewable by the

different roles mentioned above, and possibly the patient as well. It is essential that

certain parts of the document, for example, recommendations as to treatment, can be

attributed to the person that made them and can be guaranteed not to have been altered.

These needs cannot be met by TLS (previously known as SSL and described in

Section 11.6.3), which is used to create a secure channel for the communication of

information. It allows the processes at the two ends of the channel to negotiate as to the

need for authentication or encryption and the keys and algorithms to be used, both when

a channel is set up and during its lifetime. For example, data about a financial transaction

might be signed and sent in the clear until sensitive information such as credit card

details are to be given, at which point encryption will be applied.

To allow for the new type of usage outlined above, the security must be specified

within the document itself and applied to the document rather than as a property of the

channel that will convey it from one user to another.

This is possible in XML or other structured document formats, in which metadata

can be used. XML tags can be used to define the properties of the data in the document.

In particular, XML security depends on new tags that can be used to indicate the

beginning and end of sections of encrypted or signed data and of signatures. Once the

necessary security has been applied within a document, it may be sent to a variety of

different users, even by means of multicast.

Basic requirements • XML security should provide at least the same level of protection

as TLS. That is:

To be able to encrypt either an entire document or just some selected parts of it: For

example, consider the information about a financial transaction, which includes a

person’s name, the type of transaction and details about the credit or debit card being

used. In one case, just the card details could be hidden, making it possible to identify

the transaction before decrypting the record. In another case, the type of transaction

could also be hidden, so that outsiders cannot tell whether it is, for example, an order

or a payment.

To be able to sign either an entire document or just some selected parts of it: When

a document is intended to be used for cooperative work by a group of people, there

can be some critical parts of the document that should be signed in order to guarantee

that they were made by a particular person or that they have not been changed. But it

is also useful to be able to have other parts that can be altered during the use of the

document – these should not be signed.424 CHAPTER 9 WEB SERVICES

Additional basic requirements • Further requirements arise from the need to store

documents, possibly to modify them and then to send them on to a variety of different

recipients:

To add to a document that is already signed and to sign the result: For example, Alice

may sign a document and pass it on to Bob, who ‘witnesses her signature’ by adding

a remark to that effect and then signing the entire document. (Section 11.1 introduces

the names, including Alice and Bob, used for the protagonists in security protocols.)

To authorize different users to view different parts of a document: In the case of a

medical record, a researcher can view some particular section of the medical data, an

administrator can view personal details and a doctor can view both.

To add to a document that already contains encrypted sections and to encrypt part of

the new version, possibly including some of the already encrypted sections.

The flexibility and structuring capabilities of XML notation make it possible do all of

the above, without any additions to the scheme derived from the basic requirements.

Requirements concerning algorithms • XML secure documents are signed and/or encrypted well in advance of any consideration as to who will be accessing them. If the

originator is no longer involved, it is not possible to negotiate the protocols and whether

to use authentication or encryption. Therefore:

The standard should specify a suite of algorithms to be provided in any implementation of XML security: At least one encryption and one signature algorithm should be

mandatory, to enable the widest possible interoperability. Other optional algorithms

should be provided for use within smaller groups.

The algorithms used for encryption and authentication of a particular document must

be selected from that suite and the names of the algorithms in use must be referenced

within the XML document itself: If the places where the document will be used cannot

be predicted, then one of the required protocols should be used.

XML security defines the names of elements that can be used to specify the URI of the

algorithm in use for signing or encryption. So as to be able to select a variety of

algorithms within the same XML document, an element that specifies an algorithm is

generally nested inside an element containing signed information or encrypted data.

Requirements for finding keys • When a document is created and each time that it is

updated, appropriate keys must be chosen, without any negotiation with those parties

that may access the document in the future. This leads to the following requirements:

To help the users of secure documents with finding the necessary keys: For example,

a document that includes signed data should contain information as to the public key

to be used to validate the signature, such as a name that can be used to obtain the key,

or a certificate. A KeyInfo element can be used for this purpose.

To make it possible for cooperating users to help one another with keys: Provided

that the KeyInfo element is not cryptographically bound to the signature itself,

information may be added without breaking the digital signature. For example,

suppose Alice signs a document and sends it to Bob with a KeyInfo element that

specifies only the name of the key. When Bob receives the document he retrieves the

information needed to validate the signature and adds this to the KeyInfo element

when he passes the document to Carol.SECTION 9.5 XML SECURITY 425

The KeyInfo element • XML security specifies a KeyInfo element for indicating the key

to be used to validate a signature or to decrypt some data. It may contain, for example,

certificates, the names of keys or key agreement algorithms. Its use is optional: the

signer may not want to reveal any key information to all of the parties that access the

document, and in some cases the application using XML security may already have

access to the keys in use.

Canonical XML • Some applications may make changes that have no effect on the

actual information content of an XML document. This arises because there are a variety

of different ways of representing what is logically the same XML document. For

example, attributes may be in different orders and differing character encodings may be

used, yet the information content is equivalent. Canonical XML [www.w3.org X] was

designed for use with digital signatures, which are used to guarantee that the information

content of a document has not been changed. XML elements are canonicalized before

being signed and the name of the canonicalization algorithm is stored, together with the

signature. This enables the same algorithm to be used when the signature is validated.

The canonical form is a standard serialization of XML as a stream of bytes. It adds

default attributes and removes superfluous schemas, putting the attributes and schema

declarations in lexicographic order in each element. It uses a standard form for line

breaks and the UTF-8 encoding for characters. Any two equivalent XML documents

have the same canonical form.

When a subset of an XML document – say an element – is canonicalized, the

canonical form includes the ancestor context, that is, the namespaces declared and the

values of the attributes. Thus when canonical XML is used in conjunction with digital

signatures, the signature of an element will not pass its validation if that element is

placed in a different context.

A variation of this algorithm, called Exclusive Canonical XML, omits the context

from the serialization. This could be used if the application intends a particular signed

element to be used in different contexts.

Use of digital signatures in XML • The specification for digital signatures in XML

[www.w3.org XII] is a W3C recommendation that defines new XML element types to

hold signatures, the names of algorithms, keys and references to signed information. The

names provided in this specification are defined in the XML Signature schema which

Figure 9.16 Algorithms required for XML signature

Type of algorithm Name of algorithm Required reference

Message digest SHA-1 Required Section 11.4.3

Encoding base64 Required [Freed and Borenstein 1996]

Signature DSA with SHA-1 Required [NIST 1994]

(asymmetric) RSA with SHA-1 Recommended Section 11.3.2

MAC signature

(symmetric)

HMAC-SHA-1 Required Section 11.4.2 and

Krawczyk et al. [1997]

Canonicalization Canonical XML Required Page 425426 CHAPTER 9 WEB SERVICES

includes the elements Signature, SignatureValue, SignedInfo and KeyInfo. Figure 9.16

shows the algorithms that must be available in an implementation of XML Signature.

Key management service • The specification of the XML key management service

[www.w3.org XIII] contains protocols for distributing and registering public keys for

use in XML signatures. Although it does not require any particular public key

infrastructure, the service is designed to be compatible with existing ones, for example,

X.509 certificates (Section 11.4.4), SPKI (the Simple Public Key Infrastructure, Section

11.4.4) or PGP key identifiers (Pretty Good Privacy, Section 11.5.2).

Clients can use this service to find the public key of a person. For example, if Alice

wants to send an encrypted email to Bob, she can use this service to obtain his public

key. In another example, Bob receives a signed document from Alice containing her

X.509 certificate and then asks the key information service to extract the public key.

XML encryption • The standard for encryption in XML is defined in a W3C

recommendation that specifies both the way to represent encrypted data in XML and the

process for encrypting and decrypting it [www.w3.org XIV]. It introduces an

EncryptedData element for enclosing portions of encrypted data.

Figure 9.17 specifies the encryption algorithms that should be included in an

implementation of XML encryption. Block cipher algorithms are used for encrypting the

data, and base64 encoding is used in XML for representing digital signatures and

encrypted data. Key transport algorithms are public key encryption algorithms designed

for use in encrypting and decrypting the keys themselves.

Symmetric key wrap algorithms are shared secret key encryption algorithms

designed for encrypting and decrypting symmetric keys by means of another key. This

could be used if a key were to be included in a KeyInfo element.

A key agreement algorithm allows a shared secret key to be derived from a

computation on a pair of public keys. This algorithm is made available for use by

applications that need to agree a shared key without any exchange. It is not applied by

the XML security system itself.

Figure 9.17 Algorithms required for XML encryption (the algorithms in Figure 9.16 are also required)

Type of algorithm Name of algorithm Required reference

Block cipher TRIPLEDES,

AES-128, AES-256

Required Section 11.3.1

AES-192 Optional

Encoding base64 Required [Freed and Borenstein 1996]

Key transport RSA-v1.5,

RSA-OAEP

Required Section 11.3.2

[Kaliski and Staddon 1998]

Symmetric key wrap

(signature by

shared key)

TRIPLEDES

KeyWrap,

AES-128 KeyWrap,

AES-256KeyWrap

Required [Housley 2002]

AES-192 KeyWrap Optional

Key agreement Diffie-Hellman Optional [Rescorla, 1999]SECTION 9.6 COORDINATION OF WEB SERVICES 427

9.6 Coordination of web services

The SOAP infrastructure supports single request-response interactions between clients

and web services. However, many useful applications involve several requests that need

to be done in a particular order. For example, when booking a flight, the price and

availability information is collected before the reservations are made. When a user

interacts with web pages by means of a browser, for example, to book a flight or to make

a bid in an auction, the interface provided by the browser (which is based on the

information provided by the server) controls the sequence in which the operations are

performed.

However, if it is a web service that is making reservations, like the travel agent

service shown in Figure 9.2, that web service needs to work from a description of the

appropriate way to proceed when interacting with other services are used for, e.g., car

hire and hotel bookings as well as flight bookings. Figure 9.18 shows an example of such

a description.

This example illustrates the need for web services as clients to be provided with a

description of a particular protocol to follow when interacting with other web services.

But there is also the issue of maintaining consistency in the server data when it is

receiving and responding to requests from multiple clients. Chapters 16 and 17 discuss

transactions, illustrating the issues by means of a series of banking transactions. As a

Figure 9.18 Travel agent scenario

1. The client asks the travel agent service for information about a set of services; for

example, flights, car hire and hotel bookings.

2. The travel agent service collects prices and availability information and sends it

to the client, which chooses one of the following on behalf of the user:

(a) Refine the query, possibly involving more providers to get more information,

then repeat step 2.

(b) Make reservation.

(c) Quit.

3. The client requests a reservation and the travel agent service checks availability.

4. Either all are available;

or for services that are not available;

either alternatives are offered to the client, which goes back to step 3;

or the client goes back to step 1.

5. Take deposit.

6. Give the client a reservation number as a confirmation.

7. During the period until the final payment, the client may modify or cancel

reservations.428 CHAPTER 9 WEB SERVICES

simple example, in a transfer of money between two bank accounts, consistency requires

that both the deposit in one account and the withdrawal from the other must be

performed. Chapter 17 presents the two-phase commit protocol that is used by

cooperating servers to ensure consistency of transactions.

In some cases, atomic transactions suit the requirements of applications using web

services. However, activities such as those of the travel agent take a long time to

complete, and it would be impractical to use a two-phase commit protocol to carry them

out because it involves keeping resources locked for long periods of time. An alternative

is to use a more relaxed protocol in which each participant makes changes to persistent

state as they occur. In the case of failure, an application-level protocol is used to undo

these actions.

In conventional middleware, the infrastructure provides a simple request-reply

protocol, leaving other services such as transactions, persistency and security to be

implemented as separate higher-level services that can be used when they are needed.

The same is true for web services, where the W3C and others have been putting in effort

towards the definition of higher-level services.

Work has been done on a general model for coordination of web services, which

is similar to the distributed transaction model described in Section 17.2 in that it has

coordinator and participant roles that are able to act out particular protocols, for

example, to carry out a distributed transaction. This work, which is called WSCoordination, is described by Langworthy [2004]. The same group has also shown how

transactions may be carried out within this model. For a comprehensive study of web

services coordination protocols, see Alonso et al. [2004].

In the remainder of this section, we outline the ideas behind web service

choreography. Consider the fact that it would be possible to describe all of the possible

valid alternative paths through the set of interactions between pairs of web services

working together in a joint task such as the travel agent scenario. If such a description

were available, it could be used as an aid to the coordination of joint tasks. It could also

be used as a specification to be followed by new instances of a service, such as a new

flight booking service wishing to join a collaboration.

The W3C uses the term choreography to refer to a language based on WSDL for

defining coordination. For example, the language might specify constraints on the order

and the conditions in which messages are exchanged by participants. A choreography is

intended to provide a global description of a set of interactions, showing the behaviour

of each member of a set of participants, with a view to enhancing interoperability.

Requirements for choreography • Choreography is intended to support interactions

between web services which are generally managed by different companies and

organizations. A collaboration involving multiple web services and clients should be

described in terms of the sets of observable interactions between pairs of them. Such a

description might be seen as a contract between the participants, and could be used for

the following purposes:

• to generate code outlines for a new service that wants to participate;

• as a basis for generating test messages for a new service;

• to promote a common understanding of the collaboration;

• to analyze the collaboration, for example to identify possible deadlock situations.SECTION 9.7 APPLICATIONS OF WEB SERVICES 429

The use of a common choreography description by a set of collaborating web services

should result in more robust services with better interoperability. In addition, it should

be easier to develop and to introduce new services, making the overall service more

useful.

The W3C working draft document [www.w3.org XV] suggests that a

choreography language should include the following features:

• hierarchical and recursive composition of choreographies;

• the ability to add new instances of an existing service and new services;

• concurrent paths, alternative paths and the ability to repeat a section of a

choreography;

• variable timeouts – for example, different periods for holding reservations;

• exceptions, for example, to deal with messages arriving out of sequence and user

actions such as cancellations;

• asynchronous interactions (callbacks);

• reference passing, for example, to allow a car hire company to consult a bank for

a credit check on behalf of a user;

• marking of the boundaries of the separate transactions that take place, for

example, to allow for recovery;

• the ability to include human-readable documentation.

A model based on these requirements is described in another W3C working draft

document [www.w3.org XVI].

Languages for choreography • The intention is to produce a declarative, XML-based

language for defining choreographies that can make use of WSDL definitions. The W3C

has made a recommendation for Web Services Choreography Definition Language

Version 1 [www.w3.org XVII]. Prior to this, a group of companies submitted to the

W3C a specification for the web services choreography interface [www.w3.org XVIII].

9.7 Applications of web services

Web services are now one of the dominant paradigms for programming distributed

systems. In this section, we discuss a number of the major areas where web services have

been employed extensively: in supporting service-oriented architecture, the Grid and

latterly, cloud computing.

9.7.1 Service-oriented architecture

Service-oriented architecture (SOA) is a set of design principles whereby distributed

systems are developed using sets of loosely coupled services that can be dynamically

discovered and then communicate with each other or are coordinated through

choreography to provide enhanced services. Service-oriented architecture is an abstract430 CHAPTER 9 WEB SERVICES

concept that can be implemented using a variety of technologies including the

distributed object or component-based approaches discussed in Chapter 8. The principal

means of realizing service-oriented architecture, however, is through the use of web

services, largely due to the loose coupling inherent in this approach (as discussed in

Section 9.2).

This style of architecture can be used within a business or organization to offer a

flexible software architecture and to achieve interoperability between the various

services. Its prime use, however is in the broader Internet, offering a common view of

services making them globally accessible and amenable to subsequent composition.

This makes it possible to transcend the levels of heterogeneity inherent in the Internet

and also to deal with the problem of different organizations adopting different

middleware products internally – it is possible for one organization to use CORBA

internally and another to use .NET but both then to expose interfaces using web services,

thus encouraging global interoperability. The resultant property is known as businessto-business (B2B) integration. We already saw one example of the need for B2B

integration in Figure 9.18 (the travel agent scenario), where the travel agent may deal

with a wide range of companies offering flights, car rentals and hotel accommodation.

Service-oriented architecture also enables and encourages a mashup approach to

software development. A mashup is a new service created by a third-party developer by

combining two or more services available in the distributed environment. The mashup

culture relies on the ready availability of useful services with well-defined interfaces

coupled with an open innovation community where individuals or groups engage in the

development of experimental combined services and make them available to others for

further development. Both conditions are now met by the Internet, particularly with the

emergence of cloud computing and software as a service (as introduced in Section

7.7.1), where major software developers such as Amazon, Flickr and eBay make

services available through published interfaces to other developers. As an example, refer

to JBidwatcher [www.jbidwatcher.org], a Java-based mashup that interfaces to eBay to

manage bids proactively on behalf of a client, for example tracking auctions and bidding

at the last minute to maximize chances of success.

9.7.2 The Grid

The name ‘Grid’ is used to refer to middleware that is designed to enable the sharing of

resources such as files, computers, software, data and sensors on a very large scale. The

resources are shared typically by groups of users in different organizations who are

collaborating on the solution of problems requiring large numbers of computers to solve

them, either by the sharing of data or by the sharing of computing power. These

resources are necessarily supported by heterogeneous computer hardware, operating

systems, programming languages and applications. Management is needed to coordinate

the use of resources to ensure that clients get what they need and that services can afford

to supply it. In some cases, sophisticated security techniques are required to ensure that

the correct use is made of resources in this type of environment. For an example of a

Grid application, refer to the box on page 431, which features the World-Wide

Telescope application developed at Microsoft Research.SECTION 9.7 APPLICATIONS OF WEB SERVICES 431

The World-Wide Telescope: A Grid application

This project is concerned with deploying the data resources shared by the

astronomy community. It is described in the work of Szalay and Gray [2004], Szalay

and Gray [2001] and Gray and Szalay [2002]. Astronomy data consists of archives of

observations, each of which covers a particular period of time, a part of the

electromagnetic spectrum (optical, x-ray, radio) and a particular area of the sky.

These observations are made by different instruments deployed at various places

throughout the world.

A study of how astronomers share their data is useful for deriving the

characteristics of a typical Grid application, because astronomers freely share their

results with one another and issues of security can be omitted, making this discussion

simpler.

Astronomers make studies that need to combine data on the same celestial

objects but involve several different periods of time and multiple parts of the

spectrum. The ability to use independent observations of data is important to

research. Visualization allows astronomers to see the data as 2D or 3D scatter plots.

The teams gathering the data store it in immense archives (currently terabytes),

which are managed locally by each team that gathers data. The instruments used in

gathering data are subject to Moore’s law, so the amount of data gathered grows

exponentially. As it is gathered, the data is analyzed by a pipeline process and stored

as derived data for use by astronomers throughout the world. But before data can be

used by other researchers, scientists working in a particular field need to agree on a

common way of labelling their data.

Szalay and Gray [2004] point out that in the past, scientific research data was

included by authors in articles and published in journals that lived in libraries. But

nowadays, the quantity of data is too great to be included in a publication. This

applies not only to astronomy, but also to the fields of particle physics and genome

and biology research. The role of author now belongs to the collaborations, which

take 5–10 years to build their experiments before producing the data that is published

to the world in web-based archives. Thus, the scientists working on the projects

become data publishers and librarians as well as authors.

This additional role requires any project that manages a data archive to make it

accessible to other researchers. This implies a considerable overhead in addition to

the original task of data analysis. To make such sharing possible, the raw data

requires metadata to describe, for example, the time it was collected, the part of the

sky and the instrument used. In addition, the derived data needs to be accompanied

by metadata describing the parameters of the pipelines through which it was

processed.

The calculation of derived data requires heavy computational support. It often

has to be recalculated as techniques improve. All of this is a considerable expense for

the project that owns the data.

The aim of the World-Wide Telescope is to unify the world’s astronomy

archives into a giant database containing astronomy literature, images, raw data,

derived datasets and simulation data.432 CHAPTER 9 WEB SERVICES

Requirements of Grid applications • The World-Wide Telescope is typical of a range of

data-intensive Grid applications, wherein:

• data is collected by means of scientific instruments;

• the data is stored in archives at separate sites whose locations can be in different

places throughout the world;

• the data is managed by teams of scientists belonging to separate organizations;

• an immense and increasing quantity (terabytes or petabytes) of raw data is

generated from the instruments;

• computer programs are used to analyze and make summaries of the raw data, for

example, to classify, calibrate and catalogue the raw data representing celestial

objects.

The Internet makes all of these data archives potentially available to scientists

throughout the world, enabling them to get data from different instruments gathered at

different times and at different sites. However, a particular scientist using this data for

their own research will be interested in just a subset of the objects in the archives.

The immense quantity of data in an archive makes it infeasible to transfer it to the

location of the user before processing it to extract the objects of interest, due to

considerations such as transmission time and the local disk space required. Therefore, it

is not appropriate to use FTP or web access in this context. The processing of the raw

data should take place at the location where it is collected and stored in a database. Then

when a scientist makes a query about particular objects, the information in each database

should be analyzed and if necessary, visualizations produced before returning the results

to the remote query.

The fact that data is processed at many different sites provides an inbuilt

parallelism that effectively divides the immense task being undertaken.

From the above characteristics, the following requirements are derived:

R1: Remote access to resources – that is, to the required information in the archives.

R2: Processing of data at the site where it is stored and managed, either when it is

gathered or in response to a request. A typical query might result in a

visualization based on data collected for one region of sky recorded by different

instruments at different times. It will involve selecting a small quantity of data

from each massive data archive.

R3: The resource manager of a data archive should be able to create service instances

dynamically to deal with the particular section of data required, just as in the

distributed object model, where servants are created whenever they are needed

to handle different resources managed by a service.

R4: Metadata to describe:

– characteristics of the data in an archive – for example, for astronomy, the area

of the sky, the date and time collected and the instruments used;

– characteristics of a service managing that data – for example, its cost, its

geographic location, its publisher or its load or space available.SECTION 9.7 APPLICATIONS OF WEB SERVICES 433

R5: Directory services based on the above metadata.

R6: Software to manage queries, data transfers and advance reservation of resources,

taking into account that the resources are generally managed by the projects that

generate the data and that access to them may need to be rationed.

Web services can deal with the first two requirements by providing a convenient way

for scientists to access operations on data in remote archives. This will require that each

particular application provide a service description that includes a set of methods for

accessing its data. The Grid middleware must deal with the remaining requirements.

Grids are also used for computationally intensive Grid applications such as

processing the vast quantities of data produced by the CMS high-energy particle accelerator at CERN [www.uscms.org], testing the effects of candidate drug molecules

[Taufer et al. 2003, Chien 2004] or supporting massively multiplayer online games using spare capacity in cluster computers [www.butterfly.net]. Where computationally-intensive applications are deployed on a Grid, resource management will be concerned

with allocating computing resources and balancing loads.

Finally, security will be needed for many Grid applications. For example, the Grid

is in use for medical research and for business applications. Even when the privacy of

data is not an issue, it will be important to establish the identity of the people who

created the data.

Grid middleware • The Open Grid Services Architecture (OGSA) is a standard for

Grid-based applications [Foster et al. 2001, 2002]. It provides a framework within

which the above requirements can be met, based on web services. Resources are

managed by application-specific Grid services. The Globus toolkit then implements the

architecture.

The Globus Project started in 1994 with a view to providing software that

integrates and standardizes the functions required by a family of scientific applications.

These functions include directory services, security and resource management. The first

Globus toolkit appeared in 1997. The OGSA evolved from the second version of the

toolkit (called GT2), which is described in Foster and Kesselman [2004]. The third

version (GT3), which appeared in 2002, was based on OGSA and therefore built on web

services. It was developed by the Globus Alliance (www.globus.org) and is described in

Sandholm and Gawor [2003]. Since then, two further versions have been released – the

latest version is referred to as GT5 and is available as open source software

[www.globus.org]).

A case study of OGSA and the Globus toolkit (up to GT3) can be found on the

companion web site [www.cdk5.net/web].

9.7.3 Cloud computing

Cloud computing was introduced in Chapter 1 as a set of Internet-based application,

storage and computing services sufficient to support most users’ needs, thus enabling

them to largely or totally dispense with local data storage and application software.

Cloud computing also promotes a view of everything as a service, from physical or

virtual infrastructure through to software, often paid for on a per-usage basis rather than

purchased. The concept is therefore intrinsically linked to a new business model for434 CHAPTER 9 WEB SERVICES

computing where cloud suppliers offer a range of computational, data and other services

to customers as required for their daily use, for example offering sufficient storage

capacity across the Internet to act as an archival or backup service.

Chapter 1 also comments on the overlap between cloud computing and the Grid.

The development of the Grid preceded the emergence of cloud computing and was a

significant factor in its emergence. They share the same goal of providing resources

(services) out there in the greater Internet. Whereas the Grid tends to focus on high-end

data-heavy or computationally expensive applications, cloud computation is more

general, offering a range of services for individual computer users through to high-end

users. The business model associated with cloud computing is also a distinguishing

characteristic. It is therefore fair to say that the Grid is an early example of cloud

computing, but cloud computing has developed significantly since then.

With the view of everything as a service, web services offer a natural

implementation path for cloud computing, and indeed many vendors go down this path.

The most notable offering in this space is Amazon Web Services (AWS)

[aws.amazon.com], and we look briefly at this technology below. We will see an

alternative approach to cloud computing in Chapter 21, when we look at Google

infrastructure and the associated Google App Engine, which both feature a lighterweight, higher-performance approach than web services.

Amazon Web Services are a set of cloud services implemented on the extensive

physical infrastructure owned by Amazon.com. Originally developed for internal

purposes in support of their electronic retail business, Amazon now offers many of the

facilities to external users, enabling them to run independent services on the

infrastructure. The implementation of AWS takes care of key distributed systems issues

such as managing service availability, scalability and performance, allowing developers

to focus on the use of their services. Services are made available using web service

standards described earlier in this chapter. This has the advantage that programmers

familiar with web services can readily use AWS and can develop mashups that

incorporate Amazon Web Services in their construction. More generally, the approach

enables interoperability across the Internet. Amazon also adopts the REST approach, as

advocated by Fielding [2000] and discussed in Section 9.2.

Amazon offers a wide and extensible set of services, the most significant of which

are listed in Figure 9.19. We feature EC2 in more detail. EC2 is an elastic compute

service, where the term ‘elastic’ refers to the ability to offer computing capacity that is

resizeable to the customers needs. Rather than an actual machine, EC2 offers the user a

virtual machine, called an instance, to their desired specification. For example, a user

can request an instance of the following types:

• a standard instance designed to be suitable for most applications;

• a high-memory instance that offers additional memory capacity, for example for

applications involving caching;

• a high-CPU instance designed to support computationally intensive tasks;

• a cluster compute instance offering a cluster of virtual processors with highbandwidth interconnection for high-performance computing tasks.SECTION 9.8 SUMMARY 435

Several of these can be further refined – for example, for a standard instance it is

possible to request a small, medium or large instance representing different

specifications in terms of processing power, memory, disk storage and so on.

EC2 is built on top of the Xen hypervisor, described in Section 7.7.2. The

instances can be configured to run a variety of operating systems including Windows

Server 2008, Linux or OpenSolaris. They can also be configured with a variety of

software. For example, it is possible to request an installation of Apache HTTP to

support web hosting.

EC2 supports the interesting concept of an elastic IP address, which looks like a

traditional IP address but is associated with the user’s account, not a particular instance.

This means that if a (virtual) machine fails, the IP address can be reassigned to a

different machine without requiring the intervention of a network administrator.

9.8 Summary

In this chapter we have shown that web services have arisen from the need to provide an

infrastructure to support interworking between different organizations. This infrastructure generally uses the widely used HTTP protocol to transport messages between clients and servers over the Internet and is based on the use of URIs to refer to resources.

XML, a textual format, is used for data representation and marshalling.

Two separate influences led to the emergence of web services. One of these was

the addition of service interfaces to web servers with a view to allowing the resources

on a site to be accessed by client programs other than browsers and using a richer form

Figure 9.19 A selection of Amazon Web Services

Web service Description

Amazon Elastic Compute Cloud (EC2) Web-based service offering access to virtual

machines of a given performance and storage

capacity

Amazon Simple Storage Service (S3) Web-based storage service for unstructured data

Amazon Simple DB Web-based storage service for querying

structured data

Amazon Simple Queue Service (SQS) Hosted service supporting message queuing (as

discussed in Chapter 6)

Amazon Elastic MapReduce Web-based service for distributed computation

using the MapReduce model (introduced in

Chapter 21)

Amazon Flexible Payments Service (FPS) Web-based service supporting electronic

payments436 CHAPTER 9 WEB SERVICES

of interaction. The other was the desire to provide something like RPC over the Internet,

based on the existing protocols. The resulting web services provide interfaces with sets

of operations that can be called remotely. Like any other form of service, a web service

can be the client of another web service, thus allowing a web service to integrate or

combine a set of other web services.

SOAP is the communication protocol that is generally used by web services and

their clients. It can be used to transmit request messages and their replies between client

and server, either by the asynchronous exchange of documents or by a form of requestreply protocol based on a pair of asynchronous message exchanges. In both cases, the

request or reply message is enclosed in an XML-formatted document called an

envelope. The SOAP envelope is generally transmitted over the synchronous HTTP

protocol, although other transports can be used.

XML and SOAP processors are available for all of the widely used programming

languages and operating systems. This enables web services and their clients to be

deployed almost anywhere. This form of interworking is enabled by the facts that web

services are not tied to any particular programming language and do not support the

distributed object model.

In conventional middleware services, interface definitions provide clients with the

details of services. However, in the case of web services, service descriptions are used.

A service description specifies the communication protocol to be used (for example,

SOAP) and the URI of the service, as well as describing its interface. The interface may

be described either as a set of operations or as a set of messages to be exchanged between

client and server.

XML security was designed to provide the necessary protection for the contents

of a document exchanged by members of a group of people, who have different tasks to

perform on that document. Different parts of the document will be available to different

people, some with the ability to add to or alter the content and others only to read it. To

enable complete flexibility in its future use, the security properties are defined within the

document itself. This is achieved by means of XML, which is a self-describing format.

XML elements are used to specify document parts that are encrypted or signed as well

as details of the algorithms used and information to help with finding keys.

Web services have been used for a variety of purposes in distributed systems. For

example, web services provide a natural implementation of the concept of serviceoriented architecture, in which their loose coupling enables interoperability in Internetscale applications – including business-to-business (B2B) applications. Their inherent

loose coupling also supports the emergence of a mashup approach to web service

construction. Web services also underpin the Grid, supporting collaborations between

scientists or engineers in organizations in different parts of the world. Their work is very

often based on the use of raw data collected by instruments at different sites and then

processed locally. The Globus toolkit is an implementation of the architecture that has

been used in a variety of data-intensive and computationally intensive applications.

Finally, web services are heavily used in cloud computing. For example Amazon’s

AWS is based entirely on web service standards coupled with the REST philosophy of

service construction.EXERCISES 437

EXERCISES

9.1 Compare the request-reply protocol as described in Section 5.2 with the implementation

of client-server communication in SOAP. State two reasons why the use of

asynchronous messages by SOAP is more appropriate over the Internet. To what extent

does the use of HTTP by SOAP reduce the difference between the two approaches?

page 404

9.2 Compare the structure of URLs as used for web services with that of remote object

references as specified in Section 4.3.4. State in each case how they are used to execute

a client request. page 409

9.3 Illustrate the contents of a SOAP Request message and corresponding Reply message in

the Election service example of Exercise 5.11, using the pictorial version of XML as

shown in Figure 9.4 and Figure 9.5. page 405

9.4 Outline the five main elements of a WSDL service description. In the case of the

Election service defined in Exercise 5.11, state the type of information to be used by the

Request and Reply messages – does any of this need to be included in the target

namespace? For the vote operation, draw diagrams similar to Figure 9.11 and Figure

9.13. page 418

9.5 Continuing with the example of the Election service, explain why the part of the WSDL

service description defined in Exercise 9.4 is referred to as ‘abstract’. What would need

to be added to the service description to make it completely concrete? page 416

9.6 Define a Java interface for the Election service suitable for use as a web service. State

why you think the interface you defined is suitable. Explain how a WSDL document for

the service is generated and how it is made available to clients. page 412

9.7 What are the different tools used to generate the skeleton class and the service

description? Explain how these tools are used for generating the service description.

page 411

9.8 The servlet container includes a dispatcher and skeletons. Explain the role of the

dispatcher in a servlet container. page 412

9.9 Which Java API is used to hide the details of SOAP from the programmers of both

clients and services? Name the different types in the Java language that are mapped by

this Java API to definitions in XML used in SOAP messages and service descriptions.

page 410

9.10 Outline the replication scheme used in UDDI. Supposing that vector timestamps are

used to support this scheme, define a pair of operations for use by registries needing to

exchange data. page 422

9.11 Explain the purpose of UDDI. What are the four data structures that support UDDI?

Describe their uses.

pages 420, 421438 CHAPTER 9 WEB SERVICES

9.12 What are the different APIs for query operations that are provided by UDDI for looking

up services? What is the output of these query operations?

page 422

9.13 Documents protected by XML security may be signed or encrypted long before anyone

can predict who will be the ultimate recipients. What measures are taken to ensure that

the latter have access to the algorithms used by the former? page 422

9.14 Explain the relevance of canonical XML to digital signatures. What contextual

information can be included in the canonical form? Give an example of a breach of

security where the context is omitted from the canonical form. page 425

9.15 A coordination protocol could be carried out in order to coordinate the actions of web

services. Outline an architecture for (i) a centralized and (ii) a distributed coordination

protocol. In each case, describe the interactions needed to establish coordination

between a pair of web services. page 427

9.16 Compare RPC call semantics with the semantics of WS-ReliableMessaging:

i) State the entities to which each refers.

ii) Compare the differing meanings of the available semantics (for example, atleast-once, at-most-once, exactly-once).

Chapter 5 and page 408439

10

PEER-TO-PEER SYSTEMS

10.1 Introduction

10.2 Napster and its legacy

10.3 Peer-to-peer middleware

10.4 Routing overlays

10.5 Overlay case studies: Pastry, Tapestry

10.6 Application case studies: Squirrel, OceanStore, Ivy

10.7 Summary

Peer-to-peer systems represent a paradigm for the construction of distributed systems

and applications in which data and computational resources are contributed by many

hosts on the Internet, all of which participate in the provision of a uniform service. Their

emergence is a consequence of the very rapid growth of the Internet, embracing many

millions of computers and similar numbers of users requiring access to shared resources.

A key problem for peer-to-peer systems is the placement of data objects across

many hosts and subsequent provision for access to them in a manner that balances the

workload and ensures availability without adding undue overheads. We describe several

recently developed systems and applications that are designed to achieve this.

Peer-to-peer middleware systems are emerging that have the capacity to share

computing resources, storage and data present in computers ‘at the edges of the Internet’

on a global scale. They exploit existing naming, routing, data replication and security

techniques in new ways to build a reliable resource-sharing layer over an unreliable and

untrusted collection of computers and networks.

Peer-to-peer applications have been used to provide file sharing, web caching,

information distribution and other services, exploiting the resources of tens of thousands

of machines across the Internet. They are at their most effective when used to store very

large collections of immutable data. Their design diminishes their effectiveness for

applications that store and update mutable data objects.440 CHAPTER 10 PEER-TO-PEER SYSTEMS

10.1 Introduction

The demand for services in the Internet can be expected to grow to a scale that is limited

only by the size of the world’s population. The goal of peer-to-peer systems is to enable

the sharing of data and resources on a very large scale by eliminating any requirement

for separately managed servers and their associated infrastructure.

The scope for expanding popular services by adding to the number of the

computers hosting them is limited when all the hosts must be owned and managed by

the service provider. Administration and fault recovery costs tend to dominate. The

network bandwidth that can be provided to a single server site over available physical

links is also a major constraint. System-level services such as Sun NFS (Section 12.3),

the Andrew File System (Section 12.4) or video servers (Section 20.6.1) and

application-level services such as Google, Amazon or eBay all exhibit this problem to

varying degrees.

Peer-to-peer systems aim to support useful distributed services and applications

using data and computing resources available in the personal computers and

workstations that are present in the Internet and other networks in ever-increasing

numbers. This is increasingly attractive as the performance difference between desktop

and server machines narrows and broadband network connections proliferate.

But there is another, broader aim: one author [Shirky 2000] has defined peer-topeer applications as ‘applications that exploit resources available at the edges of the

Internet – storage, cycles, content, human presence’. Each type of resource sharing

mentioned in that definition is already represented by distributed applications available

for most types of personal computer. The purpose of this chapter is to describe some

general techniques that simplify the construction of peer-to-peer applications and

enhance their scalability, reliability and security.

Traditional client-server systems manage and provide access to resources such as

files, web pages or other information objects located on a single server computer or a

small cluster of tightly coupled servers. With such centralized designs, few decisions are

required about the placement of the resources or the management of server hardware

resources, but the scale of the service is limited by the server hardware capacity and

network connectivity. Peer-to-peer systems provide access to information resources

located on computers throughout a network (whether it be the Internet or a corporate

network). Algorithms for the placement and subsequent retrieval of information objects

are a key aspect of the system design. The aim is to deliver a service that is fully

decentralized and self-organizing, dynamically balancing the storage and processing

loads between all the participating computers as computers join and leave the service.

Peer-to-peer systems share these characteristics:

• Their design ensures that each user contributes resources to the system.

• Although they may differ in the resources that they contribute, all the nodes in a

peer-to-peer system have the same functional capabilities and responsibilities.

• Their correct operation does not depend on the existence of any centrally

administered systems.SECTION 10.1 INTRODUCTION 441

• They can be designed to offer a limited degree of anonymity to the providers and

users of resources.

• A key issue for their efficient operation is the choice of an algorithm for the

placement of data across many hosts and subsequent access to it in a manner that

balances the workload and ensures availability without adding undue overheads.

Computers and network connections owned and managed by a multitude of different

users and organizations are necessarily volatile resources; their owners do not guarantee

to keep them switched on, connected and fault-free. So the availability of the processes

and computers participating in peer-to-peer systems is unpredictable. Peer-to-peer

services therefore cannot rely on guaranteed access to individual resources, although

they can be designed to make the probability of failure to access a copy of a replicated

object arbitrarily small. It is worth noting that this weakness of peer-to-peer systems can

be turned into a strength if the replication of resources that it calls for is exploited to

achieve a degree of resistance to tampering by malicious nodes (for example, through

Byzantine fault-tolerance techniques; see Chapter 18).

Several early Internet-based services, including DNS (Section 13.2.3) and

Netnews/Usenet [Kantor and Lapsley 1986], adopted a multi-server scalable and faulttolerant architecture. The Xerox Grapevine name registration and mail delivery service

[Birrell et al. 1982, Schroeder et al. 1984] provides an interesting early example of a

scalable, fault-tolerant distributed service. Lamport’s part-time parliament algorithm for

distributed consensus [Lamport 1989], the Bayou replicated storage system (see Section

18.4.2) and the classless interdomain IP routing algorithm (see Section 3.4.3) are all

examples of distributed algorithms for the placement or location of information and can

be considered as antecedents of peer-to-peer systems.

But the potential for the deployment of peer-to-peer services using resources at the

edges of the Internet emerged only when a significant number of users had acquired

always-on, broadband connections to the network, making their desktop computers

suitable platforms for resource sharing. This occurred first in the United States around

1999. By mid-2004 the worldwide number of broadband Internet connections had

comfortably exceeded 100 million [Internet World Stats 2004].

Three generations of peer-to-peer system and application development can be

identified. The first generation was launched by the Napster music exchange service

[OpenNap 2001], which we describe in the next section. A second generation of filesharing applications offering greater scalability, anonymity and fault tolerance quickly

followed including Freenet [Clarke et al. 2000, freenetproject.org], Gnutella, Kazaa

[Leibowitz et al. 2003] and BitTorrent [Cohen 2003].

Peer-to-peer middleware • The third generation is characterized by the emergence of

middleware layers for the application-independent management of distributed resources

on a global scale. Several research teams have now completed the development,

evaluation and refinement of peer-to-peer middleware platforms and demonstrated or

deployed them in a range of application services. The best-known and most fully

developed examples include Pastry [Rowstron and Druschel 2001], Tapestry [Zhao et

al. 2004], CAN [Ratnasamy et al. 2001], Chord [Stoica et al. 2001] and Kademlia

[Maymounkov and Mazieres 2002].442 CHAPTER 10 PEER-TO-PEER SYSTEMS

These platforms are designed to place resources (data objects, files) on a set of

computers that are widely distributed throughout the Internet and to route messages to

them on behalf of clients, relieving clients of any need to make decisions about placing

resources and to hold information about the whereabouts of the resources they require.

Unlike the second-generation systems, they provide guarantees of delivery for requests

in a bounded number of network hops. They place replicas of resources on available host

computers in a structured manner, taking account of their volatile availability, their

variable trustworthiness and requirements for load balancing and locality of information

storage and use.

Resources are identified by globally unique identifiers (GUIDs), usually derived

as a secure hash (described in Section 11.4.3) from some or all of the resource’s state.

The use of a secure hash makes a resource ‘self certifying’ – clients receiving a resource

can check the validity of the hash. This protects it against tampering by untrusted nodes

on which it may be stored, but this technique requires that the states of resources are

immutable, since a change to the state would result in a different hash value. Hence peerto-peer storage systems are inherently best suited to the storage of immutable objects

Figure 10.1 Distinctions between IP and overlay routing for peer-to-peer applications

IP Application-level routing overlay

Scale IPv4 is limited to 232 addressable

nodes. The IPv6 namespace is much

more generous (2128), but addresses

in both versions are hierarchically

structured and much of the space is

preallocated according to

administrative requirements.

Peer-to-peer systems can address

more objects. The GUID namespace

is very large and flat (>2128),

allowing it to be much more fully

occupied.

Load balancing Loads on routers are determined by

network topology and associated

traffic patterns.

Object locations can be randomized

and hence traffic patterns are

divorced from the network topology.

Network dynamics

(addition/deletion of

objects/nodes)

IP routing tables are updated

asynchronously on a best-effort basis

with time constants on the order of 1

hour.

Routing tables can be updated

synchronously or asynchronously

with fractions-of-a-second delays.

Fault tolerance Redundancy is designed into the IP

network by its managers, ensuring

tolerance of a single router or

network connectivity failure. n-fold

replication is costly.

Routes and object references can be

replicated n-fold, ensuring tolerance

of n failures of nodes or connections.

Target identification Each IP address maps to exactly one

target node.

Messages can be routed to the nearest

replica of a target object.

Security and

anonymity

Addressing is only secure when all

nodes are trusted. Anonymity for the

owners of addresses is not

achievable.

Security can be achieved even in

environments with limited trust. A

limited degree of anonymity can be

provided.SECTION 10.1 INTRODUCTION 443

(such as music or video files). Their use for objects with changing values is more

challenging, but this can be accommodated by the addition of trusted servers to manage

a sequence of versions and identify the current version (as is done for example in

OceanStore and Ivy, described in Sections 10.6.2 and 10.6.3).

The use of peer-to-peer systems for applications that demand a high level of

availability for the objects stored requires careful application design to avoid situations

in which all of the replicas of an object are simultaneously unavailable. There is a risk

of this for objects stored on computers with the same ownership, geographic location,

administration, network connectivity, country or jurisdiction. The use of randomly

distributed GUIDs assists by distributing the object replicas to randomly located nodes

in the underlying network. If the underlying network spans many organizations across

the globe, then the risk of simultaneous unavailability is much reduced.

Overlay routing versus IP routing • At first sight, routing overlays share many

characteristics with the IP packet routing infrastructure that constitutes the primary

communication mechanism of the Internet (see Section 3.4.3). It is therefore legitimate

to ask why an additional application-level routing mechanism is required in peer-to-peer

systems. The answer lies in several distinctions that are identified in Figure 10.1. It may

be argued that some of these distinctions arise from the ‘legacy’ nature of IP as the

Internet’s primary protocol, but the legacy’s impact is too strong for it to be overcome

in order to support peer-to-peer applications more directly.

Distributed computation • The exploitation of spare computing power on end-user

computers has long been a subject of interest and experiment. Work with the first

personal computers at Xerox PARC [Shoch and Hupp 1982] showed the feasibility of

performing loosely coupled compute-intensive tasks by running background processes

on ~100 personal computers linked by a local network. More recently, much larger

numbers of computers have been put to use to perform several scientific calculations

that require almost unlimited quantities of computing power.

The most widely known effort of this type is the SETI@home project [Anderson

et al. 2002], which is part of a wider project called the Search for Extra-Terrestrial

Intelligence. SETI@home partitions a stream of digitized radio telescope data into 107-

second work units, each of about 350 kbytes and distributes them to client computers

whose computing power is contributed by volunteers. Each work unit is distributed

redundantly to 3–4 personal computers to guard against erroneous or malicious nodes

and is examined for significant signal patterns. The distribution of work units and the

coordination of results is handled by a single server that is responsible for

communication with all of the clients. Anderson et al. [2002] reported that 3.91 million

personal computers had participated in the SETI@home project by August 2002,

resulting in the processing of 221 million work units and representing an average 27.36

teraflops of computational power during the 12 months to July 2002. The work

completed to that date represented the largest single computation on record.

The SETI@home computation is unusual in that it does not involve any

communication or coordination between computers while they are processing the work

units; the results are communicated to a central server in a single short message that may

be delivered whenever the client and server are available. Some other scientific tasks of

this nature have been identified, including the search for large prime numbers and

attempts at brute-force decryption, but the unleashing of the computational power in the444 CHAPTER 10 PEER-TO-PEER SYSTEMS

Internet for a broader range of tasks will depend upon the development of a distributed

platform that supports data sharing and the coordination of computation between

participating computers on a large scale. That is the goal of the Grid project, discussed

in Chapter 19.

In this chapter we focus on algorithms and systems developed to date for the

sharing of data in peer-to-peer networks. In Section 10.2 we summarize Napster’s

design and review the lessons learned from it. In Section 10.3 we describe the general

requirements for peer-to-peer middleware layers. The following sections cover the

design and application of peer-to-peer middleware platforms, starting with an abstract

specification in Section 10.4, followed by detailed descriptions of two fully developed

examples in Section 10.5 and some applications of them in Section 10.6.

10.2 Napster and its legacy

The first application in which a demand for a globally scalable information storage and

retrieval service emerged was the downloading of digital music files. Both the need for

and the feasibility of a peer-to-peer solution were first demonstrated by the Napster filesharing system [OpenNap 2001] which provided a means for users to share files.

Napster became very popular for music exchange soon after its launch in 1999. At its

peak, several million users were registered and thousands were swapping music files

simultaneously.

Napster’s architecture included centralized indexes, but users supplied the files,

which were stored and accessed on their personal computers. Napster’s method of

operation is illustrated by the sequence of steps shown in Figure 10.2. Note that in step 5

clients are expected to add their own music files to the pool of shared resources by

Figure 10.2 Napster: peer-to-peer file sharing with a centralized, replicated index

Napster server

1. File location Index

2. List of peers

request

offering the file

peers

3. File request

4. File delivered

5. Index update

Napster server

IndexSECTION 10.2 NAPSTER AND ITS LEGACY 445

transmitting a link to the Napster indexing service for each available file. Thus the

motivation for Napster and the key to its success was the making available of a large,

widely distributed set of files to users throughout the Internet, fulfilling Shirky’s dictum

by providing access to ‘shared resources at the edges of the Internet’.

Napster was shut down as a result of legal proceedings instituted against the

operators of the Napster service by the owners of the copyright in some of the material

(i.e., digitally encoded music) that was made available on it (see the box below).

Anonymity for the receivers and the providers of shared data and other resources

is a concern for the designers of peer-to-peer systems. In systems with many nodes, the

routing of requests and results can be made sufficiently tortuous to conceal their source

and the contents of files can be distributed across multiple nodes, spreading the

responsibility for making them available. Mechanisms for anonymous communication

that are resistant to most forms of traffic analysis are available [Goldschlag et al. 1999].

If files are also encrypted before they are placed on servers, the owners of the servers

can plausibly deny any knowledge of the contents. But these anonymity techniques add

to the cost of resource sharing, and recent work has shown that the anonymity available

is weak against some attacks [Wright et al. 2002].

The Freenet [Clarke et al. 2000] and FreeHaven [Dingledine et al. 2000] projects

are focused on providing Internet-wide file services that offer anonymity for the

providers and users of the shared files. Ross Anderson has proposed the Eternity Service

[Anderson 1996], a storage service that provides long-term guarantees of data

Peer-to-peer systems and copyright ownership issues

The developers of Napster argued that they were not liable for the infringement of the

copyright owners’ rights because they were not participating in the copying process,

which was performed entirely between users’ machines. Their argument failed

because the index servers were deemed an essential part of the process. Since the

index servers were located at well-known addresses, their operators were unable to

remain anonymous and so could be targeted in lawsuits.

A more fully distributed file-sharing service might have achieved a better

separation of legal responsibilities, spreading the responsibility across all of the users

and thus making the pursuit of legal remedies very difficult, if not impossible.

Whatever view one takes about the legitimacy of file copying for the purpose of

sharing copyright-protected material, there are legitimate social and political

justifications for the anonymity of clients and servers in some application contexts.

The most persuasive justification arises when anonymity is used to overcome

censorship and maintain freedom of expression for individuals in oppressive

societies or organizations.

It is known that email and web sites have played a significant role in achieving

public awareness at times of political crisis in such societies; their role could be

strengthened if the authors could be protected by anonymity. ‘Whistle-blowing’ is a

related case: a ‘whistle-blower’ is an employee who publicizes or reports their

employer’s wrongdoings to authorities without revealing their own identity for fear

of sanctions or dismissal. In some circumstances it is reasonable for such an action

to be protected by anonymity.446 CHAPTER 10 PEER-TO-PEER SYSTEMS

availability through resistance to all sorts of accidental data loss and denial of service

attacks. He bases the need for such a service on the observation that whereas publication

is a permanent state for printed information – it is virtually impossible to delete material

once it has been published and distributed to a few thousand libraries in diverse

organizations and jurisdictions around the world – electronic publications cannot easily

achieve the same level of resistance to censorship or suppression. Anderson covers the

technical and economic requirements to ensure the integrity of the store and also points

out that anonymity is often an essential requirement for the persistence of information,

since it provides the best defence against legal challenges, as well as illegal actions such

as bribes or attacks on the originators, owners or keepers of the data.

Lessons learned from Napster • Napster demonstrated the feasibility of building a

useful large-scale service that depends almost wholly on data and computers owned by

ordinary Internet users. To avoid swamping the computing resources of individual users

(for example, the first user to offer a chart-topping song) and their network connections,

Napster took account of network locality – the number of hops between the client and

the server – when allocating a server to a client requesting a song. This simple loaddistribution mechanism enabled the service to scale to meet the needs of large numbers

of users.

Limitations: Napster used a (replicated) unified index of all available music files. For the

application in question, the requirement for consistency between the replicas was not

strong, so this did not hamper performance, but for many applications it would

constitute a limitation. Unless the access path to the data objects is distributed, object

discovery and addressing are likely to become a bottleneck.

Application dependencies: Napster took advantage of the special characteristics of the

application for which it was designed in other ways:

• Music files are never updated, avoiding any need to make sure all the replicas of

files remain consistent after updates.

• No guarantees are required concerning the availability of individual files – if a

music file is temporarily unavailable, it can be downloaded later. This reduces the

requirement for dependability of individual computers and their connections to

the Internet.

10.3 Peer-to-peer middleware

A key problem in the design of peer-to-peer applications is providing a mechanism to

enable clients to access data resources quickly and dependably wherever they are

located throughout the network. Napster maintained a unified index of available files for

this purpose, giving the network addresses of their hosts. Second-generation peer-topeer file storage systems such as Gnutella and Freenet employ partitioned and

distributed indexes, but the algorithms used are specific to each system.

This location problem existed in several services that predate the peer-to-peer

paradigm as well. For example, Sun NFS addresses this need with the aid of a virtual file

system abstraction layer at each client that accepts requests to access files stored onSECTION 10.3 PEER-TO-PEER MIDDLEWARE 447

multiple servers in terms of virtual file references (i.e., v-nodes, see Section 12.3). This

solution relies on a substantial amount of preconfiguration at each client and manual

intervention when file distribution patterns or server provision changes. It is clearly not

scalable beyond a service managed by a single organization. AFS (Section 12.4) has

similar properties.

Peer-to-peer middleware systems are designed specifically to meet the need for

the automatic placement and subsequent location of the distributed objects managed by

peer-to-peer systems and applications.

Functional requirements • The function of peer-to-peer middleware is to simplify the

construction of services that are implemented across many hosts in a widely distributed

network. To achieve this it must enable clients to locate and communicate with any

individual resource made available to a service, even though the resources are widely

distributed amongst the hosts. Other important requirements include the ability to add

new resources and to remove them at will and to add hosts to the service and remove

them. Like other middleware, peer-to-peer middleware should offer a simple

programming interface to application programmers that is independent of the types of

distributed resource that the application manipulates.

Non-functional requirements • To perform effectively, peer-to-peer middleware must

also address the following non-functional requirements [cf. Kubiatowicz 2003]:

Global scalability: One of the aims of peer-to-peer applications is to exploit the

hardware resources of very large numbers of hosts connected to the Internet. Peer-topeer middleware must therefore be designed to support applications that access

millions of objects on tens of thousands or hundreds of thousands of hosts.

Load balancing: The performance of any system designed to exploit a large number

of computers depends upon the balanced distribution of workload across them. For

the systems we are considering, this will be achieved by a random placement of

resources together with the use of replicas of heavily used resources.

Optimization for local interactions between neighbouring peers: The ‘network distance’ between nodes that interact has a substantial impact on the latency of individual interactions, such as client requests for access to resources. Network traffic

loadings are also impacted by it. The middleware should aim to place resources close

to the nodes that access them the most.

Accommodating to highly dynamic host availability: Most peer-to-peer systems are

constructed from host computers that are free to join or leave the system at any time.

The hosts and network segments used in peer-to-peer systems are not owned or

managed by any single authority; neither their reliability nor their continuous

participation in the provision of a service is guaranteed. A major challenge for peerto-peer systems is to provide a dependable service despite these facts. As hosts join

the system, they must be integrated into the system and the load must be redistributed

to exploit their resources. When they leave the system whether voluntarily or

involuntarily, the system must detect their departure and redistribute their load and

resources.

Studies of peer-to-peer applications and systems such as Gnutella and Overnet

have shown a considerable turnover of participating hosts [Saroiu et al. 2002,

Bhagwan et al. 2003]. For the Overnet peer-to-peer file-sharing system, with 85,000448 CHAPTER 10 PEER-TO-PEER SYSTEMS

active hosts throughout the Internet, Bhagwan et al. measured an average session

length of 135 minutes (and a median of 79 minutes) for a random sample of 1,468

hosts over a 7-day period, with 260 to 650 of the 1,468 hosts available to the service

at any time. (A session represents a period during which a host is available before it

is voluntarily or unavoidably disconnected.)

On the other hand, Microsoft researchers measured a session length of 37.7

hours for a random sample of 20,000 machines connected to the Microsoft corporate

network, with between 14,700 and 15,600 of the machines available for service at

any given time [Castro et al. 2003]. These measurements are based on a feasibility

study for the Farsite peer-to-peer file system [Bolosky et al. 2000]. The huge variance

amongst the figures obtained in these studies is mainly attributable to the differences

in behaviour and network environment between individual Internet users and the

users in a corporate network such as Microsoft’s.

Security of data in an environment with heterogeneous trust: In global-scale systems with participating hosts of diverse ownership, trust must be built up by the use

of authentication and encryption mechanisms to ensure the integrity and privacy of

information.

Anonymity, deniability and resistance to censorship: We have noted (in the box on

page 445) that anonymity for the holders and recipients of data is a legitimate concern

in many situations demanding resistance to censorship. A related requirement is that

the hosts that hold data should be able to plausibly deny responsibility for holding or

supplying it. The use of large numbers of hosts in peer-to-peer systems can be helpful

in achieving these properties.

How best to design a middleware layer to support global-scale peer-to-peer systems is

therefore a difficult problem. The requirements for scalability and availability make it

Figure 10.3 Distribution of information in a routing overlay

Object:

Node:

D

C’s routing knowledge

A’s routing knowledge D’s routing knowledge

B’s routing knowledge

C

A

BSECTION 10.4 ROUTING OVERLAYS 449

infeasible to maintain a database at all client nodes giving the locations of all the

resources (objects) of interest.

Knowledge of the locations of objects must be partitioned and distributed

throughout the network. Each node is made responsible for maintaining detailed

knowledge of the locations of nodes and objects in a portion of the namespace as well

as a general knowledge of the topology of the entire namespace (Figure 10.3). A high

degree of replication of this knowledge is necessary to ensure dependability in the face

of the volatile availability of hosts and intermittent network connectivity. In the systems

we describe below, replication factors as high as 16 are typically used.

10.4 Routing overlays

The development of middleware that meets the functional and non-functional

requirements outlined in the previous section is an active area of research, and several

significant middleware systems have already emerged. In this chapter we describe

several of them in detail.

In peer-to-peer systems a distributed algorithm known as a routing overlay takes

responsibility for locating nodes and objects. The name denotes the fact that the

middleware takes the form of a layer that is responsible for routing requests from any

client to a host that holds the object to which the request is addressed. The objects of

interest may be placed at and subsequently relocated to any node in the network without

client involvement. It is termed an overlay since it implements a routing mechanism in

the application layer that is quite separate from any other routing mechanisms deployed

at the network level such as IP routing. This approach to the management and location

of replicated objects was first analyzed and shown to be effective for networks involving

sufficiently many nodes in a groundbreaking paper by Plaxton et al. [1997].

The routing overlay ensures that any node can access any object by routing each

request through a sequence of nodes, exploiting knowledge at each of them to locate the

destination object. Peer-to-peer systems usually store multiple replicas of objects to

ensure availability. In that case, the routing overlay maintains knowledge of the location

of all the available replicas and delivers requests to the nearest ‘live’ node (i.e. one that

has not failed) that has a copy of the relevant object.

The GUIDs used to identify nodes and objects are an example of the ‘pure’ names

referred to in Section 13.1.1. These are also known as opaque identifiers, since they

reveal nothing about the locations of the objects to which they refer.

The main task of a routing overlay is the following:

Routing of requests to objects: A client wishing to invoke an operation on an object

submits a request including the object’s GUID to the routing overlay, which routes

the request to a node at which a replica of the object resides.

But the routing overlay must also perform some other tasks:

Insertion of objects: A node wishing to make a new object available to a peer-to-peer

service computes a GUID for the object and announces it to the routing overlay,

which then ensures that the object is reachable by all other clients.450 CHAPTER 10 PEER-TO-PEER SYSTEMS

Deletion of objects: When clients request the removal of objects from the service the

routing overlay must make them unavailable.

Node addition and removal: Nodes (i.e., computers) may join and leave the service.

When a node joins the service, the routing overlay arranges for it to assume some of

the responsibilities of other nodes. When a node leaves (either voluntarily or as a

result of a system or network fault), its responsibilities are distributed amongst the

other nodes.

An object’s GUID is computed from all or part of the state of the object using a function

that delivers a value that is, with very high probability, unique. Uniqueness is verified

by searching for another object with the same GUID. A hash function (such as SHA-1,

see Section 11.4) is used to generate the GUID from the object’s value. Because these

randomly distributed identifiers are used to determine the placement of objects and to

retrieve them, overlay routing systems are sometimes described as distributed hash

tables (DHT). This is reflected by the simplest form of API used to access them, as

shown in Figure 10.4. With this API, the put() operation is used to submit a data item to

be stored together with its GUID. The DHT layer takes responsibility for choosing a

location for it, storing it (with replicas to ensure availability) and providing access to it

via the get() operation.

A slightly more flexible form of API is provided by a distributed object location

and routing (DOLR) layer, as shown in Figure 10.5. With this interface objects can be

stored anywhere and the DOLR layer is responsible for maintaining a mapping between

object identifiers (GUIDs) and the addresses of the nodes at which replicas of the objects

are located. Objects may be replicated and stored with the same GUID at different hosts,

and the routing overlay takes responsibility for routing requests to the nearest available

replica.

With the DHT model, a data item with GUID X is stored at the node whose GUID

is numerically closest to X and at the r hosts whose GUIDs are next-closest to it

numerically, where r is a replication factor chosen to ensure a very high probability of

availability. With the DOLR model, locations for the replicas of data objects are decided

outside the routing layer and the DOLR layer is notified of the host address of each

replica using the publish() operation.

Figure 10.4 Basic programming interface for a distributed hash table (DHT) as implemented by the

PAST API over Pastry

put(GUID, data)

Stores data in replicas at all nodes responsible for the object identified by GUID.

remove(GUID)

Deletes all references to GUID and the associated data.

value = get(GUID)

Retrieves the data associated with GUID from one of the nodes responsible for it.SECTION 10.4 ROUTING OVERLAYS 451

The interfaces in Figures 10.4 and 10.5 are based on a set of abstract

representations proposed by Dabek et al. [2003] to show that most peer-to-peer routing

overlay implementations developed to date provide very similar functionality.

Research on the design of routing overlay systems began in 2000 and had reached

fruition by 2005, with the development and evaluation of several successful prototypes.

The evaluations demonstrated that their performance and dependability were adequate

for use in many production environments. In the next section we describe two of these

in detail: Pastry, which implements a distributed hash table API similar to the one

presented in Figure 10.4, and Tapestry, which implements an API similar to that shown

in Figure 10.5. Both Pastry and Tapestry employ a routing mechanism known as prefix

routing to determine routes for the delivery of messages based on the values of the

GUIDs to which they are addressed. Prefix routing narrows the search for the next node

along the route by applying a binary mask that selects an increasing number of

hexadecimal digits from the destination GUID after each hop. (This technique is also

employed in classless interdomain routing for IP packets, as outlined in Section 3.4.3.)

Other routing schemes have been developed that exploit different measures of

distance to narrow the search for the next hop destination. Chord [Stoica et al. 2001]

bases the choice on the numerical difference between the GUIDs of the selected node

and the destination node. CAN [Ratnasamy et al. 2001] uses distance in a d-dimensional

hyperspace into which nodes are placed. Kademlia [Maymounkov and Mazieres 2002]

uses the XOR of pairs of GUIDs as a metric for distance between nodes. Because XOR

is symmetric, Kademlia can maintain participants’ routing tables very simply; they

always receive requests from the same nodes contained in their routing tables.

GUIDs are not human-readable, so client applications must obtain the GUIDs for

resources of interest through some form of indexing service using human-readable

names or search requests. Ideally, these indexes are also stored in a peer-to-peer manner

to overcome the weaknesses of centralized indexes evidenced by Napster. But in simple

cases, such as music files or publications available for peer-to-peer download, they can

simply be indexed on web pages (cf. BitTorrent [Cohen 2003]). In BitTorrent a web

Figure 10.5 Basic programming interface for distributed object location and routing (DOLR) as

implemented by Tapestry

publish(GUID)

GUID can be computed from the object (or some part of it, e.g., its name). This

function makes the node performing a publish operation the host for the object

corresponding to GUID.

unpublish(GUID)

Makes the object corresponding to GUID inaccessible.

sendToObj(msg, GUID, [n])

Following the object-oriented paradigm, an invocation message is sent to an object

in order to access it. This might be a request to open a TCP connection for data

transfer or to return a message containing all or part of the object’s state. The final

optional parameter [n], if present, requests the delivery of the same message to n

replicas of the object.452 CHAPTER 10 PEER-TO-PEER SYSTEMS

index search leads to a stub file containing details of the desired resource, including its

GUID and the URL of a tracker – a host that holds an up-to-date list of network

addresses for providers willing to supply the file (see Chapter 20 for more details of the

BitTorrent protocol).

The foregoing description of routing overlays will probably have raised questions

in the reader’s mind about their performance and reliability. Answers to these questions

will emerge from the descriptions of practical routing overlay systems in the next

section.

10.5 Overlay case studies: Pastry, Tapestry

The prefix routing approach is adopted by both Pastry and Tapestry. Pastry is the

message routing infrastructure deployed in several applications including PAST

[Druschel and Rowstron 2001], an archival (immutable) file storage system

implemented as a distributed hash table with the API in Figure 10.4, and Squirrel, a peerto-peer web caching service described in Section 10.6.1. Pastry has a straightforward but

effective design that makes it a good first example for us to study in detail.

Tapestry is the basis for the OceanStore storage system, which we describe in

Section 10.6.2. It has a more complex architecture than Pastry because it aims to support

a wider range of locality approaches. We describe Tapestry in Section 10.5.2.

We also look at alternative unstructured approaches in Section 10.5.3, looking in

detail at the overlay style adopted by Gnutella.

10.5.1 Pastry

Pastry [Rowstron and Druschel 2001, Castro et al. 2002a, freepastry.org] is a routing

overlay with the characteristics that we outlined in Section 10.4. All the nodes and

objects that can be accessed through Pastry are assigned 128-bit GUIDs. For nodes,

these are computed by applying a secure hash function (such as SHA-1; see Section

11.4.3) to the public key with which each node is provided. For objects such as files, the

GUID is computed by applying a secure hash function to the object’s name or to some

part of the object’s stored state. The resulting GUIDs have the usual properties of secure

hash values – that is, they are randomly distributed in the range 0 to 2128–1. They

provide no clues as to the value from which they were computed, and clashes between

GUIDs for different nodes or objects are extremely unlikely. (If a clash occurs, Pastry

detects it and takes remedial action.)

In a network with N participating nodes, the Pastry routing algorithm will

correctly route a message addressed to any GUID in O(log N) steps. If the GUID

identifies a node that is currently active, the message is delivered to that node; otherwise,

the message is delivered to the active node whose GUID is numerically closest to it.

Active nodes take responsibility for processing requests addressed to all objects in their

numerical neighbourhood.

Routing steps involve the use of an underlying transport protocol (normally UDP)

to transfer the message to a Pastry node that is ‘closer’ to its destination. But note that

the closeness referred to here is in an entirely artificial space – the space of GUIDs. TheSECTION 10.5 OVERLAY CASE STUDIES: PASTRY, TAPESTRY 453

real transport of a message across the Internet between two Pastry nodes may require a

substantial number of IP hops. To minimize the risk of unnecessarily extended transport

paths, Pastry uses a locality metric based on network distance in the underlying network

(such as a hop counts or round-trip latency measurements) to select appropriate

neighbours when setting up the routing tables used at each node.

Thousands of hosts located at widely dispersed sites can participate in a Pastry

overlay. It is fully self-organizing: when new nodes join the overlay they obtain the data

needed to construct a routing table and other required state from existing members in

O(log N) messages, where N is the number of hosts participating in the overlay. In the

event of a node failure or departure, the remaining nodes can detect its absence and

cooperatively reconfigure to reflect the required changes in the routing structure in a

similar number of messages.

Routing algorithm • The full routing algorithm involves the use of a routing table at

each node to route messages efficiently, but for the purposes of explanation, we describe

the routing algorithm in two stages. The first stage describes a simplified form of the

algorithm that routes messages correctly but inefficiently without a routing table, and

Figure 10.6 Circular routing alone is correct but inefficient Based on Rowstron and Druschel [2001]

The dots depict live nodes. The space is considered as circular: node 0 is adjacent to node

(2128-1). The diagram illustrates the routing of a message from node 65A1FC to D46A1C

using leaf set information alone, assuming leaf sets of size 8 (l = 4). This is a degenerate type

of routing that would scale very poorly; it is not used in practice.

0 FFFFF....F (2128-1)

65A1FC

D13DA3

D471F1

D467C4

D46A1C454 CHAPTER 10 PEER-TO-PEER SYSTEMS

the second stage describes the full routing algorithm, which routes a request to any node

in O(log N) messages:

Stage I: Each active node stores a leaf set – a vector L (of size 2l) containing the

GUIDs and IP addresses of the nodes whose GUIDs are numerically closest on either

side of its own (l above and l below). Leaf sets are maintained by Pastry as nodes join

and leave. Even after a node failure, they will be corrected within a short time. (Fault

recovery is discussed below.) It is therefore an invariant of the Pastry system that the

leaf sets reflect a recent state of the system and that they converge on the current state

in the face of failures up to some maximum rate of failure.

The GUID space is treated as circular: GUID 0’s lower neighbour is 2128–1.

Figure 10.6 gives a view of active nodes distributed in this circular address space.

Since every leaf set includes the GUIDs and IP addresses of the current node’s

immediate neighbours, a Pastry system with correct leaf sets of size at least 2 can

route messages to any GUID trivially as follows: any node A that receives a message

M with destination address D routes the message by comparing D with its own GUID

A and with each of the GUIDs in its leaf set and forwarding M to the node amongst

them that is numerically closest to D.

Figure 10.7 First four rows of a Pastry routing table

The routing table is located at a node whose GUID begins 65A1. Digits are in hexadecimal.

The ns represent [GUID, IP address] pairs that act as node handles specifying the next hop

to be taken by messages addressed to GUIDs that match each given prefix. Grey-shaded

entries in the table body indicate that the prefix matches the current GUID up to the given

value of p: the next row down or the leaf set should be examined to find a route. Although

there are a maximum of 128 rows in the table, only log16 N rows will be populated on average

in a network with N active nodes.

p = GUID prefixes and corresponding node handles n

0 0 1 2 3 4 5 6 7 8 9 A B C D E F

n n n n n n n n n n n n n n n

1 60 61 62 63 64 65 66 67 68 69 6A 6B 6C 6F 6E 6F

n n n n n n n n n n n n n n n

2 650 651 652 653 654 655 656 657 658 659 65A 65B 65C 65D 65E 65F

n n n n n n n n n n n n n n n

3 65A0 65A1 65A2 65A3 65A4 65A5 65A6 65A7 65A8 65A9 65AA 65AB 65AC 65AD 65AE 65AF

n n n n n n n n n n n n n n nSECTION 10.5 OVERLAY CASE STUDIES: PASTRY, TAPESTRY 455

Figure 10.6 illustrates this for a Pastry system with l = 4. (In typical real

installations of Pastry, l = 8.) Based on the definition of leaf sets we can conclude that

at each step M is forwarded to a node that is closer to D than the current node and that

this process will eventually deliver M to the active node closest to D. But such a

routing scheme is clearly very inefficient, requiring ~ N/2l hops to deliver a message

in a network with N nodes.

Stage II: The second part of our explanation describes the full Pastry algorithm and

shows how efficient routing is achieved with the aid of routing tables.

Each Pastry node maintains a tree-structured routing table giving GUIDs and

IP addresses for a set of nodes spread throughout the entire range of 2128 possible

GUID values, with increased density of coverage for GUIDs numerically close to its

own.

Figure 10.7 shows the structure of the routing table for a specific node, and

Figure 10.8 illustrates the actions of the routing algorithm. The routing table is

structured as follows: GUIDs are viewed as hexadecimal values and the table

classifies GUIDs based on their hexadecimal prefixes. The table has as many rows as

there are hexadecimal digits in a GUID, so for the prototype Pastry system that we

are describing, there are 128/4 = 32 rows. Any row n contains 15 entries – one for

each possible value of the nth hexadecimal digit, excluding the value in the local

Figure 10.8 Pastry routing example Based on Rowstron and Druschel [2001]

Routing a message from node 65A1FC to D46A1C. With the aid of a well-populated routing

table the message can be delivered in ~ log16(N ) hops.

0 FFFFF....F (2128-1)

65A1FC

D13DA3

D4213F

D462BA

D471F1

D467C4

D46A1C456 CHAPTER 10 PEER-TO-PEER SYSTEMS

node’s GUID. Each entry in the table points to one of the potentially many nodes

whose GUIDs have the relevant prefix.

The routing process at any node A uses the information in its routing table R

and leaf set L to handle each request from an application and each incoming message

from another node according to the algorithm shown in Figure 10.9.

We can be sure that the algorithm will succeed in delivering M to its destination because

lines 1, 2 and 7 perform the actions described in Stage I of our description above, and

we have shown this to be a complete, although inefficient, routing algorithm. The

remaining steps are designed to use the routing table to improve the algorithm’s

performance by reducing the number of hops required.

Lines 4–5 come into play whenever D does not fall within the numeric range of

the current node’s leaf set and relevant routing table entries are available. The selection

of a destination for the next hop involves comparing the hexadecimal digits of D with

those of A (the GUID of the current node) from left to right to discover the length, p, of

their longest common prefix. This length is then used as a row offset, together with the

first non-matching digit of D as a column offset, to access the required element of the

routing table. The construction of the table ensures that this element (if not empty)

contains the IP address of a node whose GUID has p+1 prefix digits in common with D.

Line 7 is used when D falls outside the numeric range of the leaf set and there isn’t

a relevant routing table entry. This case is rare; it arises only when nodes have recently

failed and the table hasn’t yet been updated. The routing algorithm is able to proceed by

scanning both the leaf set and the routing table and forwarding M to another node whose

GUID has p matching prefix digits but is numerically closer to D. If that node is in L,

then we are following the Stage I procedure illustrated in Figure 10.6. If it is in R, then

it must be closer to D than any node in L, hence we are improving on Stage I.

Host integration • New nodes use a joining protocol in order to acquire their routing

table and leaf set contents and notify other nodes of changes they must make to their

Figure 10.9 Pastry’s routing algorithm

To handle a message M addressed to a node D (where R[p,i] is the element at column i,

row p of the routing table):

1. If (L-l < D < Ll) { // the destination is within the leaf set or is the current node.

2. Forward M to the element Li of the leaf set with GUID closest to D or the

current node A.

3. } else { // use the routing table to despatch M to a node with a closer GUID

4. Find p, the length of the longest common prefix of D and A,. and i, the (p+1)th

hexadecimal digit of D.

5. If (R[p,i]  null) forward M to R[p,i] // route M to a node with a longer common prefix.

6. else { // there is no entry in the routing table.

7. Forward M to any node in L or R with a common prefix of length p but a

GUID that is numerically closer.

}

}SECTION 10.5 OVERLAY CASE STUDIES: PASTRY, TAPESTRY 457

tables. First, the new node computes a suitable GUID (typically by applying the SHA-1

hash function to the node’s public key), then it makes contact with a nearby Pastry node.

(Here we use the term nearby to refer to network distance, i.e., a small number of

network hops or low transmission delay; see the box below.)

Suppose that the new node’s GUID is X and the nearby node it contacts has GUID

A. Node X sends a special join request message to A, giving X as its destination. A

despatches the join message via Pastry in the normal way. Pastry will route the join

message to the existing node whose GUID is numerically closest to X; let us call this

destination node Z.

A, Z and all the nodes (B, C,...) through which the join message is routed on its

way to Z add additional steps to the normal Pastry routing algorithm, which result in the

transmission of the contents of the relevant parts of their routing tables and leaf sets to

X. X examines them and constructs its own routing table and leaf set from them,

requesting some additional information from other nodes if necessary.

To see how X builds its routing table, note that the first row of the table depends

on the value of X’s GUID, and to minimize routing distances, the table should be

constructed to route messages via neighbouring nodes whenever possible. A is a

neighbour of X, so the first row of A’s table is a good initial choice for the first row of

X’s table, X0. On the other hand, A’s table is probably not relevant for the second row,

X1, because X’s and A’s GUIDs may not share the same first hexadecimal digit. The

routing algorithm ensures that X’s and B’s GUIDs do share the same first digit, though,

which implies that the second row of B’s routing table, B1, is a suitable initial value for

X1. Similarly, C2 is suitable for X2, and so on.

Furthermore, recalling the properties of leaf sets, note that since Z’s GUID is

numerically closest to X’s, X’s leaf set should be similar to Z’s. In fact, X’s ideal leaf set

will differ from Z’s by just one member. Z’s leaf set is therefore taken as an adequate

initial approximation, which will eventually be optimized through interaction with its

neighbours as described in the fault tolerance subsection below.

Finally, once X has constructed its leaf set and routing table in the manner outlined

above, it sends their contents to all the nodes identified in the leaf set and the routing

table and they adjust their own tables to incorporate the new node. The entire task of

incorporating a new node into the Pastry infrastructure requires the transmission of

O(log N) messages.

Host failure or departure • Nodes in the Pastry infrastructure may fail or depart without

warning. A Pastry node is considered failed when its immediate neighbours (in GUID

space) can no longer communicate with it. When this occurs, it is necessary to repair the

leaf sets that contain the failed node’s GUID.

Nearest neighbour algorithm

The new node should have the address of at least one existing Pastry node, but it

might not be nearby. To ensure that nearby nodes are known Pastry includes a

‘nearest neighbour’ algorithm to find a nearby node by recursively measuring the

round-trip delay for a probe message sent periodically to each member of the leaf set

of the nearest currently known Pastry node.458 CHAPTER 10 PEER-TO-PEER SYSTEMS

To repair its leaf set L, the node that discovers the failure looks for a live node

close to the failed node in L and requests a copy of that node’s leaf set, L’. L’ will contain

a sequence of GUIDs that partly overlap those in L, including one with an appropriate

value to replace the failed node. Other neighbouring nodes are then informed of the

failure and they perform a similar procedure. This repair procedure guarantees that leaf

sets will be repaired unless l adjacently numbered nodes fail simultaneously.

Repairs to routing tables are made on a ‘when discovered’ basis. The routing of

messages can proceed with some routing table entries that are no longer live – failed

routing attempts result in the use of a different entry from the same row of a routing

table.

Locality • The Pastry routing structure is highly redundant: there are many routes

between each pair of nodes. The construction of the routing tables aims to take

advantage of this redundancy to reduce actual message transmission times by exploiting

the locality properties of nodes in the underlying transport network (which is normally

a subset of nodes in the Internet).

Recall that each row in a routing table contains 16 entries. The entries in the ith

row give the addresses of 16 nodes with GUIDs with i–1 initial hexadecimal digits that

match the current node’s GUID and an ith digit that takes each of the possible

hexadecimal values. A well-populated Pastry overlay will contain many more nodes

than can be contained in an individual routing table; whenever a new routing table is

being constructed a choice is made for each position between several candidates (taken

from routing information supplied by other nodes) based on a proximity neighbour

selection algorithm [Gummadi et al. 2003]. A locality metric (number of IP hops or

measured latency) is used to compare candidates and the closest available node is

chosen. Since the information available is not comprehensive, this mechanism cannot

produce globally optimal routings, but simulations have shown that it results in routes

that are on average only about 30–50% longer than the optimum.

Fault tolerance • As described above, the Pastry routing algorithm assumes that all

entries in routing tables and leaf sets refer to live, correctly functioning nodes. All nodes

send ‘heartbeat’ messages (i.e., messages sent at fixed time intervals to indicate that the

sender is alive) to neighbouring nodes in their leaf sets, but information about failed

nodes detected in this manner may not be disseminated sufficiently rapidly to eliminate

routing errors. Nor does it account for malicious nodes that may attempt to interfere with

correct routing. To overcome these problems, clients that depend upon reliable message

delivery are expected to employ an at-least-once delivery mechanism (see Section

5.3.1) and repeat their requests several times in the absence of a response. This will

allow Pastry a longer time window to detect and repair node failures.

To deal with any remaining failures or malicious nodes, a small degree of

randomness is introduced into the route selection algorithm described in Figure 10.9.

Essentially, the step in line 5 of Figure 10.9 is modified in a randomly selected small

proportion of cases to yield a common prefix that is less than the maximum length. This

results in the use of a routing taken from an earlier row of the routing table, producing

less optimal but different routing than the standard version of the algorithm. With this

random variation in the routing algorithm, client retransmissions should eventually

succeed even in the presence of a small number of malicious nodes.SECTION 10.5 OVERLAY CASE STUDIES: PASTRY, TAPESTRY 459

Dependability • The authors of Pastry have developed an updated version called

MSPastry [Castro et al. 2003] that uses the same routing algorithm and similar host

management methods, but also includes some additional dependability measures and

some performance optimizations in the host management algorithms.

Dependability measures include the use of acknowledgements at each hop in the

routing algorithm. If the sending host does not receive an acknowledgement after a

specified timeout, it selects an alternative route and retransmits the message. The node

that failed to send an acknowledgement is then noted as a suspected failure.

As mentioned above, to detect failed nodes each Pastry node periodically sends a

heartbeat message to its immediate neighbour to the left (i.e., with a lower GUID) in the

leaf set. Each node also records the time of the last heartbeat message received from its

immediate neighbour on the right (with a higher GUID). If the interval since the last

heartbeat exceeds a timeout threshold, the detecting node starts a repair procedure that

involves contacting the remaining nodes in the leaf set with a notification about the

failed node and a request for suggested replacements. Even in the case of multiple

simultaneous failures, this procedure terminates with all nodes on the left side of the

failed node having leaf sets that contain the l live nodes with the closest GUIDs.

We have seen that the routing algorithm can function correctly using leaf sets

alone; but the maintenance of the routing tables is important for performance. Suspected

failed nodes in routing tables are probed in a similar manner to that used for the leaf set

and if they fail to respond, their routing table entries are replaced with a suitable

alternative obtained from a nearby node. In addition, a simple gossip protocol (see

Section 18.4.1) is used to periodically exchange routing table information between

nodes in order to repair failed entries and prevent slow deterioration of the locality

properties. The gossip protocol is run about every 20 minutes.

Evaluation work • Castro and his colleagues have carried out an exhaustive

performance evaluation of MSPastry, aimed at determining the impact on performance

and dependability of the host join/leave rate and the associated dependability

mechanisms [Castro et al. 2003].

The evaluation was performed by running the MSPastry system under control of

a simulator running on a single machine that simulates a large network of hosts, with

message passing replaced by simulated transmission delays. The simulation realistically

modelled the join/leave behaviour of hosts and IP transmission delays based on

parameters from real installations.

All of the dependability mechanisms of MSPastry were included, with realistic

intervals for probe and heartbeat messages. The simulation work was validated by

comparison with measurements taken with MSPastry running a real application load

across an internal network with 52 nodes.

Here we summarize the key results.

Dependability: With an assumed IP message loss rate of 0%, MSPastry failed to

deliver 1.5 in 100,000 requests (presumably due to the non-availability of destination

hosts), and all requests that were delivered arrived at the correct node.

With an assumed IP message loss rate of 5%, MSPastry lost about 3.3 in

100,000 requests and 1.6 in 100,000 requests were delivered to the wrong node. The

use of per-hop acknowledgements in MSPastry ensures that all lost or misdirected

messages are eventually retransmitted and reach the correct node.460 CHAPTER 10 PEER-TO-PEER SYSTEMS

Performance: The metric used to evaluate the performance of MSPastry is called

relative delay penalty (RDP) [Chu et al. 2000], or stretch. RDP is a direct measure

of the extra cost incurred in employing an overlay routing layer. It is the ratio between

the average delay in delivering a request by the routing overlay and in delivering a

similar message between the same two nodes via UDP/IP. The RDP values observed

for MSPastry under simulated loads ranged from ~1.8 with zero network message

loss to ~2.2 with 5% network message loss.

Overheads: The extra network load generated by control traffic – messages involved

in maintaining leaf sets and routing tables – was less than 2 messages per minute per

node. The RDP and control traffic were both increased significantly for session

lengths less than about 60 minutes due to initial setup overheads.

Overall these results show that overlay routing layers can be constructed that achieve

good performance and high dependability with thousands of nodes operating in realistic

environments. Even with mean session lengths shorter than 60 minutes and high

network error rates the system degrades gracefully, continuing to provide an effective

service.

Optimizing overlay lookup latency • Zhang et al. [2005a] have shown that the lookup

performance of an important class of overlay networks (including Pastry, Chord and

Tapestry) can be substantially enhanced by the inclusion of a simple learning algorithm

that measures the latencies actually experienced in accessing the overlay nodes and thus

incrementally modifies the overlay routing tables to optimize access latencies.

10.5.2 Tapestry

Tapestry implements a distributed hash table and routes messages to nodes based on

GUIDs associated with resources using prefix routing in a manner similar to Pastry. But

Tapestry’s API conceals the distributed hash table from applications behind a DOLR

interface like the one shown in Figure 10.5. Nodes that hold resources use the

publish(GUID) primitive to make them known to Tapestry, and the holders of resources

remain responsible for storing them. Replicated resources are published with the same

GUID by each node that holds a replica, resulting in multiple entries in the Tapestry

routing structure.

This gives Tapestry applications additional flexibility: they can place replicas

close (in network distance) to frequent users of resources in order to reduce latency and

minimize network load or to ensure tolerance of network and host failures. But this

distinction between Pastry and Tapestry is not fundamental: Pastry applications can

achieve similar flexibility by making the objects associated with GUIDs simply act as

proxies for more complex application-level objects and Tapestry can be used to

implement a distributed hash table in terms of its DOLR API [Dabek et al. 2003].

In Tapestry 160-bit identifiers are used to refer both to objects and to the nodes

that perform routing actions. Identifiers are either NodeIds, which refer to computers

that perform routing operations, or GUIDs, which refer to the objects. For any resource

with GUID G there is a unique root node with GUID RG that is numerically closest to

G. Hosts H holding replicas of G periodically invoke publish(G) to ensure that newly

arrived hosts become aware of the existence of G. On each invocation of publish(G) aSECTION 10.5 OVERLAY CASE STUDIES: PASTRY, TAPESTRY 461

publish message is routed from the invoker towards node RG. On receipt of a publish

message RG enters (G, IPH), the mapping between G and the sending host’s IP address

in its routing table, and each node along the publication path caches the same mapping.

This process is illustrated in Figure 10.10. When nodes hold multiple (G, IP) mappings

for the same GUID, they are sorted by the network distance (round-trip time) to the IP

address. For replicated objects this results in the selection of the nearest available replica

of the object as the destination for subsequent messages sent to the object.

Zhao et al. [2004] give full details of the Tapestry routing algorithms and the

management of Tapestry’s routing tables in the face of node arrival and departure. Their

paper includes comprehensive performance evaluation data based on simulation of

large-scale Tapestry networks, showing that its performance is similar to Pastry’s. In

Section 10.6.2 we describe the OceanStore file store, which has been built and deployed

over Tapestry.

10.5.3 From structured to unstructured peer-to-peer

The discussion so far has focused exclusively on what are known as structured peer-topeer approaches. In structured approaches, there is an overall global policy governing

the topology of the network, the placement of objects in the network and the routing or

search functions used to locate objects in the network. In other words, there is a specific

(distributed) data structure underpinning the associated overlay and a set of algorithms

Figure 10.10 Tapestry routing From Zhao et al. [2004]

Replicas of the file Phil’s Books (G=4378) are hosted at nodes 4228 and AA93. Node 4377 is

the root node for object 4378. The Tapestry routings shown are some of the entries in routing

tables. The publish paths show routes followed by the publish messages laying down cached

location mappings for object 4378. The location mappings are subsequently used to route

messages sent to 4378.

4228

4377

437A

4361

43FE

4664

4B4F

E791

4A6D

57EC AA93

4378

Phil’s

Books

4378

Phil’s

Books

(Root for 4378)

publish path

Tapestry routings

for 4377

Location mapping

for 4378

Routes actually

taken by send(4378)462 CHAPTER 10 PEER-TO-PEER SYSTEMS

operating over that data structure. This can clearly be seen in the examples of Pastry and

Tapestry based on the underlying distributed hash table and associated ring structures.

Because of the structure imposed, such algorithms are efficient and offer time bounds

on the location of objects, but at the cost of maintaining the underlying structures, often

in highly dynamic environments.

Because of this maintenance argument, unstructured peer-to-peer approaches

have also been developed. In unstructured approaches, there is no overall control over

the topology or the placement of objects within the network. Rather, the overlay is

created in an ad hoc manner, with each node that joins the network following some

simple, local rules to establish connectivity. In particular, a joining node will establish

contact with a set of neighbours knowing that the neighbours will also be connected to

further neighbours and so on, forming a network that is fundamentally decentralized and

self-organizing and hence resilient to node failure. To locate a given object, it is then

necessary to carry out a search of the resultant network topology; clearly, this approach

cannot offer any guarantees of being able to find the object and performance will be

unpredictable. In addition, there is a real risk of generating excessive message traffic to

locate objects.

A summary of the relative strengths of structured and unstructured peer-to-peer

systems is provided in Figure 10.11. It is interesting to reflect that, despite the apparent

drawbacks of unstructured peer-to-peer systems, this approach is dominant in the

Internet, particularly in supporting peer-to-peer file sharing (with systems such as

Gnutella, FreeNet and BitTorrent all adopting unstructured approaches). As will be

seen, significant advancements have been made in such systems to improve the

performance of unstructured approaches and this work is significant given the amount

of traffic generated by peer-to-peer file sharing in the Internet (for example, a study

carried out in the years 2008/9 indicates that peer-to-peer file-sharing applications

account for between 43% and 70% of all Internet traffic, depending on the part of the

world being considered [www.ipoque.com]).

Strategies for effective search • In peer-to-peer file sharing, all nodes in the network

offer files to the greater environment. As mentioned above, the problem of locating a file

then maps onto a search of the whole network to locate appropriate files. If implemented

naively, this would be implemented by flooding the network with requests. This is

precisely the strategy adopted in early versions of Gnutella. In particular, in Gnutella

Figure 10.11 Structured versus unstructured peer-to-peer systems

Structured peer-to-peer Unstructured peer-to-peer

Advantages Guaranteed to locate objects (assuming

they exist) and can offer time and

complexity bounds on this operation;

relatively low message overhead.

Self-organizing and naturally resilient to

node failure.

Disadvantages Need to maintain often complex

overlay structures, which can be

difficult and costly to achieve,

especially in highly dynamic

environments.

Probabilistic and hence cannot offer

absolute guarantees on locating objects;

prone to excessive messaging overhead

which can affect scalability.SECTION 10.5 OVERLAY CASE STUDIES: PASTRY, TAPESTRY 463

0.4, every node forwarded a request to each of its neighbours, who then in turn passed

this on to their neighbours, and so on until a match was found. Each search was also

constrained with a time-to-live field limiting the number of hops. At the time Gnutella

0.4 was deployed, the average connectivity was around 5 neighbours per node. This

approach is simple but does not scale and quickly floods the network with search-related

traffic.

A number of refinements have been developed for search in unstructured

networks [Lv et al. 2002, Tsoumakos and Roussopoulos 2006], including:

Expanded ring search: In this approach, the initiating node carries out a series of

searches with increasing values in the time-to-live field, recognizing that a significant

number of the requests will be met locally (especially if coupled with an effective

replication strategy, as discussed below).

Random walks: With random walks, the initiating node sets off a number of walkers

who follow their own random pathways through the interconnected graph offered by

the unstructured overlay.

Gossiping: In gossiping approaches, a node sends a request to a given neighbour with

a certain probability, and hence requests propagate through the network in a manner

similar to a virus through a population (as such, gossip protocols are also referred to

as epidemic protocols). The probability can either be fixed for a given network or

calculated dynamically based on previous experience and/or the current context.

(Note that gossiping is a common technique in distributed systems; further

applications can be found in Chapters 6 and 18).

Such strategies can significantly reduce the overhead of search in unstructured networks

and hence increase the scalability of the algorithms. Such strategies are also often

supported by appropriate replication techniques. By replicating content across a number

of peers, the probability of efficient discovery of particular files is significantly

enhanced. Techniques include whole file replication and the scattering of fragments of

files across the Internet – an approach that is used effectively in BitTorrent, for example,

to reduce the burden on any one peer in downloading large files (see Chapter 20).

Case study: Gnutella • Gnutella was originally launched in 2000 and since then has

grown to be one of the dominant and most influential peer-to-peer file-sharing

applications. As mentioned above, initially the protocol adopted a rather simple flooding

strategy that did not scale particularly well. In response to this, Gnutella 0.6 introduced

a range of modifications that have significantly improved the performance of the

protocol.

The first major amendment was to move from a pure peer-to-peer architecture

where all nodes are equal to one where all peers still cooperate to offer the service but

some nodes, designated to have additional resources, are elected as ultrapeers, and form

the heart of the network, with other peers taking on the role of leaf nodes (or leaves).

Leaves connect themselves to a small number of ultrapeers which are heavily connected

to other ultrapeers (with over 32 connections each). This dramatically reduces the

maximum number of hops required for exhaustive search. This style of peer-to-peer

architecture is referred to as a hybrid architecture and is also the approach adopted in

Skype (as discussed in Section 4.5.2).464 CHAPTER 10 PEER-TO-PEER SYSTEMS

The other key enhancement was the introduction of a Query Routing Protocol

(QRP) designed to reduce the number of queries issued by nodes. The protocol is based

on exchanging information about files contained on nodes and hence only forwarding

queries down paths where the system thinks there will be a positive outcome. Rather

than sharing information about files directly, the protocol produces a set of numbers

from hashing on the individual words in a file name. For example, a file name such as

“Chapter ten on P2P” will be represented by four numbers, say <65, 47, 09, 76>. A node

produces a Query Routing Table (QRT) containing the hash values representing each of

the files contained on that node which it then sends to all its associated ultrapeers.

Ultrapeers then produce their own Query Routing Tables based on a union of all the

entries from all the connected leaves together with entries for files contained in that

node, and exchange these with other connected ultrapeers. In this way, ultrapeers can

determine which paths offer a valid route for a given query, thus significantly reducing

the amount of unnecessary traffic. More specifically, an ultrapeer will forward a query

to a node if a match is found (indicating that node has the file) and will carry out the

same check before passing it on to another ultrapeer if it is the last hop to the file. Note

that, in order to avoid overloading of ultrapeers, nodes will send a query to one ultrapeer

at a time and then wait for a specified period to see if they get a positive response.

Finally, a query in Gnutella contains the network address of the initiating

ultrapeer, which implies that once a file is found it can be sent directly to the associated

ultrapeer (using UDP), avoiding a reverse traversal through the graph.

The major elements associated with Gnutella 0.6 are summarized in Figure 10.12.

Figure 10.12 Key elements in the Gnutella protocol

File QRT

Ultrapeer

QRT

Leaf nodeSECTION 10.6 APPLICATION CASE STUDIES: SQUIRREL, OCEANSTORE, IVY 465

10.6 Application case studies: Squirrel, OceanStore, Ivy

The routing overlay layers described in the preceding section have been exploited in

several application experiments and the resulting applications have been extensively

evaluated. We have chosen three of them for further study, the Squirrel web caching

service based on Pastry, and the OceanStore and Ivy file stores.

10.6.1 Squirrel web cache

The authors of Pastry have developed the Squirrel peer-to-peer web caching service for

use in local networks of personal computers [Iyer et al. 2002]. In medium and large local

networks web caching is typically performed using a dedicated server computer or

cluster. The Squirrel system performs the same task by exploiting storage and

computing resources already available on desktop computers in the local network. We

first give a brief general description of the operation of a web caching service, then we

outline the design of Squirrel and review its effectiveness.

Web caching • Web browsers generate HTTP GET requests for Internet objects like

HTML pages, images, etc. These may be serviced from a browser cache on the client

machine, from a proxy web cache (a service running on another computer in the same

local network or on a nearby node in the Internet) or from the origin web server (the

server whose domain name is included in the parameters of the GET request), depending

on which contains a fresh copy of the object. The local and proxy caches each contain a

set of recently retrieved objects organized for fast lookup by URL. Some objects are

uncacheable because they are generated dynamically by the server in response to each

request.

When a browser cache or proxy web cache receives a GET request, there are three

possibilities: the requested object is uncacheable, there is a cache miss or the object is

found in the cache. In the first two cases the request is forwarded to the next level

towards the origin web server. When the requested object is found in a cache, the cached

copy must be tested for freshness.

Web objects are stored in web servers and cache servers with some additional

metadata values including a timestamp giving a date of last modification (T) and

possibly a time-to-live (t) or an eTag (a hash computed from the contents of a web page).

These metadata items are supplied by the origin server whenever an object is returned

to a client.

Objects that have an associated time-to-live t, are considered fresh if T+t is later

than the current real time. For objects without a time-to-live, an estimated value for t is

used (often only a few seconds). If the result of this freshness evaluation is positive, the

cached object is returned to the client without contacting the origin web server.

Otherwise, a conditional GET (cGET) request is issued to the next level for validation.

There are two basic types of cGET requests: an If-Modified-Since request containing the

timestamp of the last known modification, and an If-None-Match request containing an

eTag representing the object contents. This cGET request can be serviced either by

another web cache or by the origin server. A web cache that receives a cGET request and

does not have a fresh copy of the object forwards the request towards the origin web466 CHAPTER 10 PEER-TO-PEER SYSTEMS

server. The response contains either the entire object, or a not-modified message if the

cached object is unchanged.

Whenever a newly modified cacheable object is received from the origin server,

it is added to the set of objects in the local cache (displacing older objects that are still

valid if necessary) together with a timestamp, a time-to-live and an eTag if available.

The scheme described above is the basis of operation for the centralized proxy

web caching services deployed in most local networks that support large numbers of

web clients. Proxy web caches are typically implemented as a multi-threaded process

running on a single dedicated host or a set of processes running on a cluster of computers

and require a substantial quantity of dedicated computing resources in both cases.

Squirrel • The Squirrel web caching service performs the same functions using a small

part of the resources of each client computer on a local network. The SHA-1 secure hash

function is applied to the URL of each cached object to produce a 128-bit Pastry GUID.

Since the GUID is not used to validate the contents, it need not be based on the entire

object contents, as it is in other Pastry applications. The authors of Squirrel base their

justification for this on the end-to-end argument (Section 2.3.3), arguing that the

authenticity of a web page may be compromised at many points in its journey from the

host to the client; authentication of cached pages adds little to any overall guarantee of

authenticity and the HTTPS protocol (incorporating end-to-end Transport Layer

Security, discussed in Section 11.6.3) should be used to achieve a much better guarantee

for those interactions that require it.

In the simplest implementation of Squirrel – which proved to be the most effective

one – the node whose GUID is numerically closest to the GUID of an object becomes

that object’s home node, responsible for holding the cached copy of the object when

there is one.

Client nodes are configured to include a local Squirrel proxy process, which takes

responsibility for both local and remote caching of web objects. If a fresh copy of a

required object is not in the local cache, Squirrel routes a Get request or a cGet request

(when there is a stale copy of the object in the local cache) via Pastry to the home node.

If the home node has a fresh copy, it directly responds to the client with a not-modified

message or a fresh copy, as appropriate. If the home node has a stale copy or no copy of

the object, it issues a cGet or a Get to the origin server, respectively. The origin server

may respond with a not-modified message or a copy of the object. In the former case, the

home node revalidates its cache entry and forwards a copy of the object to the client. In

the latter case, it forwards a copy of the new value to the client and places a copy in its

local cache if the object is cacheable.

Evaluation of Squirrel • Squirrel was evaluated by simulation using modelled loads

derived from traces of the activity of existing centralized proxy web caches in two real

working environments within Microsoft, one with 105 active clients (in Cambridge) and

the other with more than 36,000 (in Redmond). The evaluation compared the

performance of a Squirrel web cache with a centralized one in three respects:

The reduction in total external bandwidth used: The total external bandwidth is

inversely related to the hit ratio, since it is only cache misses that generate requests

to external web servers. The hit ratios observed for centralized web cache servers

were 29% (for Redmond) and 38% (for Cambridge). When the same activity logsSECTION 10.6 APPLICATION CASE STUDIES: SQUIRREL, OCEANSTORE, IVY 467

were used to generate a simulated load for the Squirrel cache, with each client

contributing 100 Mbytes of disk storage, very similar hit ratios of 28% (Redmond)

and 37% (Cambridge) were achieved. It follows that the external bandwidth would

be reduced by a similar proportion.

The latency perceived by users for access to web objects: The use of a routing

overlay results in several message transfers (routing hops) across the local network

to transmit a request from a client to the host responsible for caching the relevant

object (the home node). The mean numbers of routing hops observed in the

simulation were 4.11 hops to deliver a GET request in the Redmond case and 1.8 hops

in the Cambridge case, whereas only a single message transfer is required to access

a centralized cache service.

However local transfers take only a few milliseconds with modern Ethernet

hardware, including TCP connection setup time, whereas wide area TCP message

transfers across the Internet require 10–100 ms. The Squirrel authors therefore argue

that the latency for access to objects found in the cache is swamped by the much

greater latency of access to objects not found in the cache, giving a similar user

experience to that provided with a centralized cache.

The computational and storage load imposed on client nodes: The average number

of cache requests served for other nodes by each node over the whole period of the

evaluation was extremely low, at only 0.31 per minute (Redmond), indicating that the

overall proportion of system resources consumed is extremely low.

Based on the measurements described above, the authors of Squirrel concluded that its

performance is comparable to that of a centralized cache. Squirrel achieves a reduction

in the observed latency for web page access close to that achievable by a centralized

cache server with a similarly sized dedicated cache. The additional load imposed on

client nodes is low and likely to be imperceptible to users. The Squirrel system was

subsequently deployed as the primary web cache in a local network with 52 client

machines using Squirrel, and the results confirmed their conclusions.

10.6.2 OceanStore file store

The developers of Tapestry have designed and built a prototype for a peer-to-peer file

store. Unlike PAST, it supports the storage of mutable files. The OceanStore design

[Kubiatowicz et al. 2000; Kubiatowicz 2003; Rhea et al. 2001, 2003] aims to provide a

very large scale, incrementally scalable persistent storage facility for mutable data

objects with long-term persistence and reliability in an environment of constantly

changing network and computing resources. OceanStore is intended for use in a variety

of applications including the implementation of an NFS-like file service, electronic mail

hosting, databases and other applications involving the sharing and persistent storage of

large numbers of data objects.

The design includes provision for the replicated storage of both mutable and

immutable data objects. The mechanism for maintaining consistency between replicas

can be tailored to application needs in a manner that was inspired by the Bayou system

(Section 18.4.2). Privacy and integrity are achieved through the encryption of data and

the use of a Byzantine agreement protocol (see Section 15.5) for updates to replicated468 CHAPTER 10 PEER-TO-PEER SYSTEMS

objects. This is needed because the trustworthiness of individual hosts cannot be

assumed.

An OceanStore prototype, called Pond [Rhea et al. 2003], has been built. It is

sufficiently complete to support applications and its performance has been evaluated

against a variety of benchmarks in order to validate the OceanStore design and compare

its performance with more traditional approaches. In the remainder of this section we

give an overview of the OceanStore/Pond design and summarize the evaluation results.

Pond uses the Tapestry routing overlay mechanism to place blocks of data at

nodes distributed throughout the Internet and to despatch requests to them.

Storage organization • OceanStore/Pond data objects are analogous to files, with their

data stored in a set of blocks. But each object is represented as an ordered sequence of

immutable versions that are (in principle) kept forever. Any update to an object results

in the generation of a new version. The versions share any unchanged blocks, following

the copy-on-write technique for creating and updating objects described in Section

7.4.2. So a small difference between versions requires only a small amount of additional

storage.

Objects are structured in a manner that is reminiscent of the Unix filing system,

with the data blocks organized and accessed through a metadata block called the root

block and additional indirection blocks if necessary (cf. Unix i-nodes). Another level of

indirection is used to associate a persistent textual or other externally visible name (for

example, the pathname for a file) with the sequence of versions of a data object. Figure

Figure 10.13 Storage organization of OceanStore objects

d1 d2 d3 d4 d5

root block

version i indirection blocks

d2

version i+1

BGUID (copy on write)

d1 d3

certificate VGUID of current

version

VGUID of

version i

AGUID

Version i +1 has been updated in blocks d1, d2 and d3. The certificate and the root blocks include

some metadata not shown. All unlabelled arrows are BGUIDs.

VGUID of version i–1

data blocksSECTION 10.6 APPLICATION CASE STUDIES: SQUIRREL, OCEANSTORE, IVY 469

10.13 illustrates this organization. GUIDs are associated with the object (an AGUID),

the root block for each version of the object (a VGUID), the indirection blocks and the

data blocks (BGUIDs). Several replicas of each block are stored at peer nodes selected

according to locality and storage availability criteria, and their GUIDs are published

(using the publish() primitive of Figure 10.5) by each of the nodes that holds a replica

so that Tapestry can be used by clients to access the blocks.

Three types of GUIDs are used, as summarized in Figure 10.14. The first two are

GUIDs of the type normally assigned to objects stored in Tapestry – they are computed

from the contents of the relevant block using a secure hash function so that they can be

used later to authenticate and verify the integrity of the contents. The blocks that they

reference are necessarily immutable, since any change to the contents of a block would

invalidate the use of the GUID as an authentication token.

The third type of identifier used is AGUIDs. These refer (indirectly) to the entire

stream of versions of an object, enabling clients to access the current version of the

object or any previous version. Since the objects stored are mutable, the GUIDs used to

identify them cannot be derived from their contents, because that would render GUIDs

held in indexes, etc., obsolete whenever an object changed.

Instead, whenever a new storage object is created a permanent AGUID is

generated by applying a secure hash function to an application-specific name (e.g., a file

name) supplied by the client creating the object and a public key that represents the

object’s owner (see Section 11.2.5). In a filing system application, an AGUID would be

stored in the directories against each file name.

The association between an AGUID and the sequence of versions of the object

that it identifies is recorded in a signed certificate that is stored and replicated by a

primary copy replication scheme (also called passive replication; see Section 18.3.1).

The certificate includes the VGUID of the current version and the root block for every

version contains the VGUID of the previous version, so there is a chain of references

enabling clients that hold a certificate to traverse the entire chain of versions (Figure

10.13). A signed certificate is needed to ensure that the association is authentic and has

been made by an authorized principal. Clients are expected to check this. Whenever a

new version of an object is created, a new certificate is generated holding the VGUID

of the new version together with a timestamp and a version sequence number.

The trust model for peer-to-peer systems requires that construction of each new

certificate is agreed (as described below) amongst a small set of hosts called the inner

ring. Whenever a new object is stored in OceanStore, a set of hosts is selected to act as

the inner ring for that object. They use Tapestry’s publish() primitive to make the

AGUID for the object known to Tapestry. Clients can then use Tapestry to route requests

for the object’s certificate to one of the nodes in the inner ring.

Figure 10.14 Types of identifier used in OceanStore

Name Meaning Description

BGUID block GUID Secure hash of a data block

VGUID version GUID BGUID of the root block of a version

AGUID active GUID Uniquely identifies all the versions of an object470 CHAPTER 10 PEER-TO-PEER SYSTEMS

The new certificate replaces the old primary copy held at each inner ring node and

is disseminated to a larger number of secondary copies. It is left to clients to determine

how often they check for a new version (a similar decision has to be taken for cached

copies of files in NFS; most installations operate with a consistency window of 30

seconds between client and server; see Section 12.3).

As usual in peer-to-peer systems, trust cannot be placed in any individual host.

The updating of primary copies requires consensus agreement between the hosts in the

inner ring. They use a version of a state-machine-based Byzantine agreement algorithm

described by Castro and Liskov [2000] to update the object and sign the certificate. The

use of a Byzantine agreement protocol ensures that the certificate is correctly maintained

even if some members of the inner ring fail or behave maliciously. Because the

computational and communication costs of Byzantine agreement rise with the square of

the number of hosts involved, the number of hosts in the inner ring is kept small and the

resulting certificate is replicated more widely using the primary copy scheme mentioned

above.

Performing an update also involves checking access rights and serializing the

update with any other pending writes. Once the update process is completed for the

primary copy, the results are disseminated to secondary replicas stored on hosts outside

the inner ring using a multicast routing tree that is managed by Tapestry.

Because of their read-only nature, data blocks are replicated by a different, more

storage-efficient mechanism. This mechanism is based on the division of each block into

m equal-sized fragments, which are encoded using erasure codes [Weatherspoon and

Kubiatowicz 2002] to n fragments, where n>m. The key property of erasure coding is

that it is possible to reconstruct a block from any m of its fragments. In a system that uses

erasure coding all data objects remain available with the loss of up to n–m hosts. In the

Pond implementation m = 16 and n = 32, so for a doubling of the storage cost, the system

can tolerate the failure of up to 16 hosts without loss of data. Tapestry is used to store

fragments in and retrieve them from the network.

This high level of fault tolerance and data availability is achieved at some cost in

terms of reconstructing blocks from erasure-coded fragments. To minimize the impact

of this, the whole blocks are also stored in the network using Tapestry. Since they can

be reconstructed from their fragments, these blocks are treated as a cache – they are not

fault tolerant and they can be disposed of when storage space is required.

Performance • Pond was developed as a prototype to prove the feasibility of a scalable

peer-to-peer file service, rather than as a production implementation. It is implemented

in Java and includes almost all of the design outlined above. It was evaluated against

several purpose-designed benchmarks and in a simple emulation of an NFS client and

server in terms of OceanStore objects. The developers tested the NFS emulation against

the Andrew benchmark [Howard et al. 1988], which emulates a software development

workload. The table in Figure 10.15 shows the results for the latter. They were obtained

using 1 GHz Pentium III PC running Linux. The LAN tests were performed using a

Gigabit Ethernet and the WAN results were obtained using two sets of nodes linked by

the Internet.

The conclusions drawn by the authors were that the performance of

OceanStore/Pond when operating over a wide area network (i.e., the Internet)

substantially exceeds that of NFS for reading and is within a factor of three of NFS forSECTION 10.6 APPLICATION CASE STUDIES: SQUIRREL, OCEANSTORE, IVY 471

updating files and directories; the LAN results were substantially worse. Overall, the

results suggest that an Internet-scale peer-to-peer file service based on the OceanStore

design would be an effective solution for the distribution of files that do not change very

rapidly (such as cached copies of web pages). Its potential for use as an alternative to

NFS is questionable even for wide-area networks and is clearly uncompetitive for purely

local use.

These results were obtained with data blocks stored without erasure-code-based

fragmentation and replication. The use of public keys contributes substantially to the

computational cost of Pond’s operation. The figures shown are for 512-bit keys, whose

security is good but less than perfect. The results for 1024-bit keys were substantially

worse for the phases of those benchmarks that involved file updates. Other results

obtained with purpose-designed benchmarks included measurement of the impact of the

Byzantine agreement process on the latency of updates. These were in the range of 100

ms to 10 seconds. A test of update throughput achieved a maximum of 100

updates/second.

10.6.3 Ivy file system

Like OceanStore, Ivy [Muthitacharoen et al. 2002] is a read/write file system supporting

multiple readers and writers implemented over an overlay routing layer and a distributed

hash-addressed data store. Unlike OceanStore, the Ivy file system emulates a Sun NFS

server. Ivy stores the state of files as logs of the file update requests issued by Ivy clients

and reconstructs the files by scanning the logs whenever it is unable to satisfy an access

request from its local cache. The log records are held in the DHash distributed hashaddressed storage service [Dabek et al. 2001]. (Logs were first used to record file

updates in the Sprite distributed operating system [Rosenblum and Ooosterhout 1992],

as described briefly in Section 12.5, but there they were used simply to optimize the

update performance of the file system.)

Figure 10.15 Performance evaluation of the Pond prototype emulating NFS

The figures show times in seconds to run different phases of the Andrew benchmark. It has

five phases: (1) creates subdirectories recursively, (2) copies a source tree, (3) examines the

status of all the files in the tree without examining their data, (4) examines every byte of data

in all the files and (5) compiles and links the files.

LAN WAN Predominant

operations in

benchmark

Phase Linux

NFS

Pond Linux

NFS

Pond

1 0.0 1.9 0.9 2.8 Read and write

2 0.3 11.0 9.4 16.8 Read and write

3 1.1 1.8 8.3 1.8 Read

4 0.5 1.5 6.9 1.5 Read

5 2.6 21.0 21.5 32.0 Read and write

Total 4.5 37.2 47.0 54.9472 CHAPTER 10 PEER-TO-PEER SYSTEMS

The design of Ivy resolves several previously unresolved issues arising from the

need to host files in partially trusted or unreliable machines, including:

• The maintenance of consistent file metadata (cf. i-node contents in Unix/NFS file

systems) with potentially concurrent file updates at different nodes. Locking is not

used because the failure of nodes or network connectivity might cause indefinite

blocking.

• Partial trust between participants and vulnerability to attacks of participants’

machines. Recovery from integrity failures caused by such attacks is based on the

notion of views of the file system. A view is a representation of the state

constructed from logs of the updates made by a set of participants. Participants

may be removed and a view recomputed without their updates. Thus a shared file

system is seen as the result of merging all the updates performed by a

(dynamically selected) set of participants.

• Continued operation during partitions in the network, which can result in

conflicting updates to shared files. Conflicting updates are resolved using methods

related to those used in the Coda file system (Section 18.4.3).

Ivy implements an API at each client node that is based on the NFS server protocol

(similar to the set of operations listed in Section 12.3, Figure 12.9). Client nodes include

an Ivy server process that uses DHash to store and access log records at nodes

throughout a local or wide area network based on keys (GUIDs) that are computed as

the hash of the record contents (see Figure 10.16). DHash implements a programming

interface like the one shown in Figure 10.4 and replicates all entries at several nodes for

Figure 10.16 Ivy system architecture

DHash server

Modifled

NFS client

module

Ivy server DHash server

Application

Kernel

Ivy node

DHash server

DHash server

DHash server

ApplicationSECTION 10.6 APPLICATION CASE STUDIES: SQUIRREL, OCEANSTORE, IVY 473

resilience and availability. The Ivy authors note that DHash could in principle be

replaced by another distributed hash-addressed store such as Pastry, Tapestry or CAN.

An Ivy file store consists of a set of update logs, one log per participant. Each Ivy

participant appends only to its own log but can read from all the logs that comprise the

file system. Updates are stored in separate per-participant logs so that they can be rolled

back in case of security breaches or consistency failures.

An Ivy log is a reverse time-ordered linked list of log entries. Each log entry is a

timestamped record of a client request to change the contents or metadata of a file or

directory. DHash uses the 160-bit SHA-1 hash of a record as a key for placing and

retrieving the record. Each participant also maintains a mutable DHash block (called a

log-head) that points to the participant’s most recent log record. Mutable blocks are

assigned a cryptographic public-key pair by their owner. The contents of the block are

signed with the private key and can therefore be authenticated with the corresponding

public key. Ivy uses version vectors (that is, vector timestamps; see Section 14.4) to

impose a total order on log entries when reading from multiple logs.

DHash stores a log record using a SHA-1 hash of its contents as the key. Log

records are chained in timestamp order using the DHash key as a link. The log-head

holds the key for the most recent log entry. To store and retrieve log-heads, a public key

pair is computed by the owner of the log. The public key value is used as its DHash key

and the private key is used by the owner to sign the log-head. Any participant that has

the public key can retrieve the log-head and use it to access all of the records in the log.

Assuming a file system composed of a single log for the moment, the canonical

execution method for a request to read a sequence of bytes from a file requires a

sequential scan of the log to find the log records that contain updates for the relevant

portion of the file. Logs are of unlimited length, but the scan terminates when the first

record or records are found that cover the required sequence of bytes.

The canonical algorithm to access a multi-user, multiple-log file system involves

the comparison of vector timestamps in log records to determine the order of updates

(since a global clock cannot be assumed).

The time taken to perform this process for an operation as simple as a read request

is potentially very long. It is reduced to a more tolerable and predicable duration through

the use of a combination of local caches and snapshots. Snapshots are representations of

the file system computed and stored locally by each participant as a by-product of their

use of the logs. They constitute a soft representation of the file system in the sense that

they may be invalidated if a participant is ejected from the system.

Update consistency is close-to-open; that is, the updates performed on a file by an

application are not visible to other processes until the file is closed. The use of a closeto-open consistency model enables write operations on a file to be saved at the client

node until the file is closed; then the entire set of write operations is written as a single

log record and a new log-head record is generated and written (an extension to the NFS

protocol enables the occurrence of a close operation in the application to be notified to

the Ivy server).

Since there is a separate Ivy server at each node and each autonomously stores its

updates in a separate log without coordination with the other servers, the serialization of

updates must be done at the time when logs are read in order to construct the content of

files. The version vectors written into log records can be used to order most updates, but474 CHAPTER 10 PEER-TO-PEER SYSTEMS

conflicting updates are possible and they must be resolved by application-specific

automatic or manual methods, as is done in Coda (see Section 18.4.3).

Data integrity is achieved by a combination of the mechanisms that we have

already mentioned: log records are immutable and their address is a secure hash of their

contents; log-heads are verified by checking a public-key signature of their contents. But

the trust model allows for the possibility that a malicious participant may gain access to

a file system. For example, they might delete files that they own maliciously. When this

is detected, the malicious participant is ejected from the view; their log is no longer used

to calculate the contents of the file system and files that they have deleted are once again

visible in the new view.

The Ivy authors used a modified Andrew benchmark [Howard et al. 1988] to

compare the performance of Ivy with a standard NFS server in local and wide area

network environments. They considered (a) Ivy using local DHash servers compared to

a single local NFS server and (b) Ivy using DHash servers located at several remote

Internet sites compared to a single remote NFS server. They also considered the

performance characteristics as a function of the numbers of participants in a view, the

number of participants writing concurrently and the number of DHash servers used to

store the logs.

They found that Ivy execution times were within a factor of two of NFS execution

times for most of the tests in the benchmark and within a factor of three for all of them.

The execution times for the wide area network deployment exceeded those for the local

case by a factor of 10 or more, but similar ratios were obtained for a remote NFS server.

Full details of the performance evaluation can be found in the Ivy paper [Muthitacharoen

et al. 2002]. It should be noted, though, that NFS was not designed for wide area use;

the Andrew File System and other more recently developed server-based systems such

as xFS [Anderson et al. 1996] offer higher performance in wide area deployments and

might have made better bases for the comparison. The primary contribution of Ivy is in

its novel approach to the management of security and integrity in an environment of

partial trust – an inevitable feature of very large distributed systems that span many

organizations and jurisdictions.

10.7 Summary

Peer-to-peer architectures were first shown to support very large scale data sharing with

the Internet-wide use of Napster and its descendants for digital music sharing. The fact

that much of their use conflicted with copyright laws doesn’t diminish their technical

significance, although they did also have technical drawbacks that restricted their

deployment to applications in which guarantees of data integrity and availability were

unimportant.

Subsequent research resulted in the development of peer-to-peer middleware

platforms that deliver requests to data objects wherever they are located in the Internet.

In structured approaches, the objects are addressed using GUIDs, which are pure names

containing no IP addressing information. Objects are placed at nodes according to some

mapping function that is specific to each middleware system. Delivery is performed by

a routing overlay in the middleware that maintains routing tables and forwards requestsEXERCISES 475

along a route determined by calculating distance according to the chosen mapping

function. In unstructured approaches, nodes form themselves into an ad hoc network and

then propagate searches through neighbours to find appropriate resources. Several

strategies have been developed to improve the performance of this search function and

increase the overall scalability of the system.

The middleware platforms add integrity guarantees based on the use of a secure

hash function to generate the GUIDs and availability guarantees based on the replication

of objects at several nodes and on fault-tolerant routing algorithms.

The platforms have been deployed in several large-scale pilot applications, refined

and evaluated. Recent evaluation results indicate that the technology is ready for

deployment in applications involving large numbers of users sharing many data objects.

The benefits of peer-to-peer systems include:

• their ability to exploit unused resources (storage, processing) in the host

computers;

• their scalability to support large numbers of clients and hosts with excellent

balancing of the loads on network links and host computing resources;

• the self-organizing properties of the middleware platforms which result in support

costs that are largely independent of the numbers of clients and hosts deployed.

Weaknesses and subjects of current research include:

• their use for the storage of mutable data is relatively costly compared to a trusted,

centralized service;

• the promising basis that they provide for client and host anonymity has not yet

resulted in strong guarantees of anonymity.

EXERCISES

10.1 Explain the characteristics of a peer-to-peer service. What are some examples of peerto- peer middleware?

pages 440, 441

10.2 The problem of maintaining indexes of available resources is application-dependent.

Consider the suitability of each of your answers to Exercise 10.1 for:

i) music and media file sharing;

ii) long-term storage of archived material such as journal or newspaper content;

iii)network storage of general-purpose read-write files.

10.3 What are the non-functional requirements of peer-to-peer middleware? page 447

10.4 The guarantees offered by conventional servers may be violated as a result of:

i) physical damage to the host;

ii) Errors or inconsistencies introduced by system administrators or their managers;476 CHAPTER 10 PEER-TO-PEER SYSTEMS

iii)successful attacks on the security of the system software;

iv)hardware or software errors.

Give two examples of possible incidents for each type of violation. Which of them could

be described as a breach of trust or a criminal act? Would they be breaches of trust if

they occurred on a personal computer that was contributing some resources to a peer-topeer service? Why is this relevant for peer-to-peer systems? Section 11.1.1

10.5 Peer-to-peer systems typically depend on untrusted and volatile computer systems for

most of their resources. Trust is a social phenomenon with technical consequences.

Volatility (i.e., unpredictable availability) also is often due to human actions. Elaborate

your answers to Exercise 10.4 by discussing the possible ways in which each of them

are likely to differ according to the following attributes of the computers used:

i) ownership;

ii) geographic location;

iii)network connectivity;

iv)country or jurisdiction.

What does this suggest about policies for the placement of data objects in a peer-to-peer

storage service?

10.6 Assess the availability and trustworthiness of the personal computers in your

environment. You should estimate:

Uptime: How many hours per day is the computer operating and connected to the

Internet?

Software consistency: Is the software managed by a competent technician?

Security: Is the computer fully protected against tampering by its users or others?

Based on your assessment, discuss the feasibility of running a datasharing service on the

set of computers you have assessed and outline the problems that must be addressed in

a peer-to-peer data sharing service. pages 447–448

10.7 What are the tasks of a routing overlay? Give a comparative study between structured

and unstructured peer-to-peer systems.

pages 449, 461, 462

10.8 The design of Ivy resolves several previously unresolved issues arising from the need to

host files in partially trusted or unreliable machines. Explain these issues.

pages 471, 472

10.9 Routing algorithms choose a next hop according to an estimate of distance in some

addressing space. Pastry and Tapestry both use circular linear address spaces in which a

function based on the approximate numerical difference between GUIDs determinesEXERCISES 477

their separation. Kademlia uses the XOR of the GUIDs. How does this help in the

maintenance of routing tables? Does the XOR operation provide appropriate properties

for a distance metric? pages 451, [Maymounkov and Mazieres 2002]

10.10 When the Squirrel peer-to-peer web caching service was evaluated by simulation, 4.11

hops were required on average to route a request for a cache entry when simulating the

Redmond traffic, whereas only 1.8 were required for the Cambridge traffic. Explain this

and show that it supports the theoretical performance claimed for Pastry.

pages 452, 466

10.11 In unstructured peer-to-peer systems, significant improvements on search results can be

provided by the adoption of particular search strategies. Compare and contrast expanded

ring search and random walk strategies, highlighting when each approach is likely to be

effective. page 462This page intentionally left blank479

11

SECURITY

11.1 Introduction

11.2 Overview of security techniques

11.3 Cryptographic algorithms

11.4 Digital signatures

11.5 Cryptography pragmatics

11.6 Case studies: Needham–Schroeder, Kerberos, TLS, 802.11 WiFi

11.7 Summary

There is a pervasive need for measures to guarantee the privacy, integrity and availability

of resources in distributed systems. Security attacks take the forms of eavesdropping,

masquerading, tampering and denial of service. Designers of secure distributed systems

must cope with exposed service interfaces and insecure networks in an environment

where attackers are likely to have knowledge of the algorithms used and to deploy

computing resources.

Cryptography provides the basis for the authentication of messages as well as their

secrecy and integrity; carefully designed security protocols are required to exploit it. The

selection of cryptographic algorithms and the management of keys are critical to the

effectiveness, performance and usability of security mechanisms. Public-key

cryptography makes it easy to distribute cryptographic keys but its performance is

inadequate for the encryption of bulk data. Secret-key cryptography is more suitable for

bulk encryption tasks. Hybrid protocols such as Transport Layer Security (TLS) establish

a secure channel using public-key cryptography and then use it to exchange secret keys

for use in subsequent data exchanges.

Digital information can be signed, producing digital certificates. Certificates enable

trust to be established among users and organizations.

The chapter concludes with case studies on the approaches to security system

design and the security mechanisms deployed in Kerberos, TLS/SSL and 802.11 WiFi.480 CHAPTER 11 SECURITY

11.1 Introduction

In Section 2.4.3 we introduced a simple model for examining the security requirements

in distributed systems. We concluded that the need for security mechanisms in

distributed systems arises from the desire to share resources. (Resources that are not

shared can generally be protected by isolating them from external access.) If we regard

shared resources as objects, then the requirement is to protect any processes that

encapsulate shared objects and any communication channels that are used to interact

with them against all conceivable forms of attack. The model introduced in Section 2.4.3

provides a good starting point for the identification of security requirements. It can be

summarized as follows:

• Processes encapsulate resources (both programming language–level objects and

system-defined resources) and allow clients to access them through interfaces.

Principals (users or other processes) are authorized to operate on resources.

Resources must be protected against unauthorized access (Figure 2.17).

• Processes interact through a network that is shared by many users. Enemies

(attackers) can access the network. They can copy or attempt to read any message

transmitted through the network and they can inject arbitrary messages, addressed

to any destination and purporting to come from any source, into the network

(Figure 2.18).

The need to protect the integrity and privacy of information and other resources

belonging to individuals and organizations is pervasive in both the physical and the

digital world. It arises from the desire to share resources. In the physical world,

organizations adopt security policies that provide for the sharing of resources within

specified limits. For example, a company may permit entry to its buildings only to its

employees and accredited visitors. A security policy for documents may specify groups

of employees who can access classes of documents, or it may be defined for individual

documents and users.

Security policies are enforced with the help of security mechanisms. For example,

access to a building may be controlled by a reception clerk, who issues badges to

accredited visitors, and enforced by a security guard or by electronic door locks. Access

to paper documents is usually controlled by concealment and restricted distribution. In

the electronic world, the distinction between security policies and mechanisms is

equally important; without it, it would be difficult to determine whether a particular

system was secure. Security policies are independent of the technology used, just as the

provision of a lock on a door does not ensure the security of a building unless there is a

policy for its use (for example, that the door will be locked whenever nobody is guarding

the entrance). The security mechanisms that we describe here do not in themselves

ensure the security of a system. In Section 11.1.2, we outline the requirements for

security in various simple electronic commerce scenarios, illustrating the need for

policies in that context.

The provision of mechanisms for the protection of data and other resources in

distributed systems while allowing interactions between computers that are permitted

by security policies is the concern of this chapter. The mechanisms that we shall describe

are designed to enforce security policies against the most determined attacks.SECTION 11.1 INTRODUCTION 481

The role of cryptography • Digital cryptography provides the basis for most computer

security mechanisms, but it is important to note that computer security and cryptography

are distinct subjects. Cryptography is the art of encoding information in a format that

only the intended recipients can decode. Cryptography can also be employed to provide

proof of the authenticity of information, in a manner analogous to the use of signatures

in conventional transactions.

Cryptography has a long and fascinating history. The military need for secure

communication and the corresponding need of an enemy to intercept and decrypt it led

to the investment of much intellectual effort by some of the best mathematical brains of

their time. Readers interested in exploring this history will find absorbing reading in

books on the topic by David Kahn [Kahn 1967, 1983, 1991] and Simon Singh [Singh

1999]. Whitfield Diffie, one of the inventors of public-key cryptography, has written

with firsthand knowledge on the recent history and politics of cryptography [Diffie

1988, Diffie and Landau 1998].

It is only in recent times that cryptography has emerged from the wraps previously

placed on it by the political and military establishments that used to control its

development and use. It is now the subject of open research by a large and active

community, with the results presented in many books, journals and conferences. The

publication of Schneier’s book Applied Cryptography [Schneier 1996] was a milestone

in the opening up of knowledge in the field. It was the first book to publish many

important algorithms with source code – a courageous step, because when the first

edition appeared in 1994 the legal status of such publication was unclear. Schneier’s

book remains the definitive reference on most aspects of modern cryptography. A more

recent book co-authored by Schneier [Ferguson and Schneier 2003] provides an

excellent introduction to computer cryptography and a discursive overview of virtually

all the important algorithms and techniques in current use, including several published

since Schneier’s earlier book. In addition, Menezes et al. [1997] provide a good practical

handbook with a strong theoretical basis and the Network Security Library

[www.secinf.net] is an excellent online source of practical knowledge and experience.

Ross Anderson’s Security Engineering [Anderson 2008] is also outstanding. It is

replete with object lessons on the design of secure systems, drawn from real-world

situations and system security failures.

The new openness is largely a result of the tremendous growth of interest in nonmilitary applications of cryptography and the security requirements of distributed

computer systems, which has led to the existence for the first time of a self-sustaining

community of cryptographic researchers outside the military domain.

Ironically, the opening of cryptography to public access and use has resulted in a

great improvement in cryptographic techniques, both in their strength to withstand

attacks by enemies and in the convenience with which they can be deployed. Public-key

cryptography is one of the fruits of this openness. As another example, we note that the

DES encryption algorithm that was adopted and used by the US military and

government agencies was initially a military secret. Its eventual publication and

successful efforts to crack it resulted in the development of much stronger secret-key

encryption algorithms.

Another useful spin-off has been the development of a common terminology and

approach. An example of the latter is the adoption of a set of familiar names for

protagonists (principals) involved in the transactions that are to be secured. The use of482 CHAPTER 11 SECURITY

familiar names for principals and attackers helps to clarify and bring to life descriptions

of security protocols and potential attacks on them, which aids in identifying their

weaknesses. The names shown in Figure 11.1 are used extensively in the security

literature and we use them freely here. We have not been able to discover their origins;

the earliest occurrence of which we are aware is in the original RSA public-key

cryptography paper [Rivest et al. 1978]. An amusing commentary on their use can be

found in Gordon [1984].

11.1.1 Threats and attacks

Some threats are obvious – for example, in most types of local network it is easy to

construct and run a program on a connected computer that obtains copies of the

messages transmitted between other computers. Other threats are more subtle – if clients

fail to authenticate servers, a program might install itself in place of an authentic file

server and thereby obtain copies of confidential information that clients unwittingly

send to it for storage.

In addition to the danger of loss or damage to information or resources through

direct violations, fraudulent claims may be made against the owner of a system that is

not demonstrably secure. To avoid such claims, the owner must be in a position to

disprove the claim by showing that the system is secure against such violations or by

producing a log of all of the transactions for the period in question. A common instance

is the ‘phantom withdrawal’ problem in automatic cash dispensers (teller machines).

The best answer that a bank can supply to such a claim is to provide a record of the

transaction that is digitally signed by the account holder in a manner that cannot be

forged by a third party.

The main goal of security is to restrict access to information and resources to just

those principals that are authorized to have access. Security threats fall into three broad

classes:

Leakage: Refers to the acquisition of information by unauthorized recipients.

Tampering: Refers to the unauthorized alteration of information.

Vandalism: Refers to interference with the proper operation of a system without gain

to the perpetrator.

Figure 11.1 Familiar names for the protagonists in security protocols

Alice First participant

Bob Second participant

Carol Participant in three- and four-party protocols

Dave Participant in four-party protocols

Eve Eavesdropper

Mallory Malicious attacker

Sara A serverSECTION 11.1 INTRODUCTION 483

Attacks on distributed systems depend upon obtaining access to existing communication

channels or establishing new channels that masquerade as authorized connections. (We

use the term channel to refer to any communication mechanism between processes.)

Methods of attack can be further classified according to the way in which a channel is

misused:

Eavesdropping: Obtaining copies of messages without authority.

Masquerading: Sending or receiving messages using the identity of another

principal without their authority.

Message tampering: Intercepting messages and altering their contents before

passing them on to the intended recipient. The man-in-the-middle attack is a form of

message tampering in which an attacker intercepts the very first message in an

exchange of encryption keys to establish a secure channel. The attacker substitutes

compromised keys that enable them to decrypt subsequent messages before reencrypting them in the correct keys and passing them on.

Replaying: Storing intercepted messages and sending them at a later date. This

attack may be effective even with authenticated and encrypted messages.

Denial of service: Flooding a channel or other resource with messages in order to

deny access for others.

These are the dangers in theory, but how are attacks carried out in practice? Successful

attacks depend upon the discovery of loopholes in the security of systems.

Unfortunately, these are all too common in today’s systems, and they are not necessarily

particularly obscure. Cheswick and Bellovin [1994] identify 42 weaknesses that they

regard as posing serious risks in widely used Internet systems and components. They

range from password guessing to attacks on the programs that perform the Network

Time Protocol or handle mail transmission. Some of these have led to successful and

well-publicized attacks [Stoll 1989, Spafford 1989], and many of them have been

exploited for mischievous or criminal purposes.

When the Internet and the systems that are connected to it were designed, security

was not a priority. The designers probably had no conception of the scale to which the

Internet would grow, and the basic design of systems such as UNIX predates the advent

of computer networks. As we shall see, the incorporation of security measures needs to

be carefully thought out at the basic design stage, and the material in this chapter is

intended to provide the basis for such thinking.

We focus on the threats to distributed systems that arise from the exposure of their

communication channels and their interfaces. For many systems, these are the only

threats that need to be considered (other than those that arise from human error – security

mechanisms cannot guard against a badly chosen password or one that is carelessly

disclosed). But for systems that include mobile programs and systems whose security is

particularly sensitive to information leakage, there are further threats.

Threats from mobile code • Several recently developed programming languages have

been designed to enable programs to be loaded into a process from a remote server and

then executed locally. In that case, the internal interfaces and objects within an executing

process may be exposed to attack by mobile code.484 CHAPTER 11 SECURITY

Java is the most widely used language of this type, and its designers paid

considerable attention to the design and construction of the language and the

mechanisms for remote loading in an effort to restrict the exposure (the sandbox model

of protection against mobile code).

The Java virtual machine (JVM) is designed with mobile code in view. It gives

each application its own environment in which to run. Each environment has a security

manager that determines which resources are available to the application. For example,

the security manager might stop an application reading and writing files or give it

limited access to network connections. Once a security manager has been set, it cannot

be replaced. When a user runs a program such as a browser that downloads mobile code

to be run locally on their behalf, they have no very good reason to trust the code to

behave in a responsible manner. In fact, there is a danger of downloading and running

malicious code that removes files or accesses private information. To protect users

against untrusted code, most browsers specify that applets cannot access local files,

printers or network sockets. Some applications of mobile code are able to assume

various levels of trust in downloaded code. In this case, the security managers are

configured to provide more access to local resources.

The JVM takes two further measures to protect the local environment:

1. The downloaded classes are stored separately from the local classes, preventing

them from replacing local classes with spurious versions.

2. The bytecodes are checked for validity. Valid Java bytecode is composed of Java

virtual machine instructions from a specified set. The instructions are also checked

to ensure that they will not produce certain errors when the program runs, such as

accessing illegal memory addresses.

The security of Java has been the subject of much subsequent investigation, in the course

of which it became clear that the original mechanisms adopted were not free of

loopholes [McGraw and Felden 1999]. The identified loopholes were corrected and the

Java protection system was refined to allow mobile code to access local resources when

authorized to do so [java.sun.com V].

Despite the inclusion of type-checking and code-validation mechanisms, the

security mechanisms incorporated into mobile code systems do not yet produce the same

level of confidence in their effectiveness as those used to protect communication

channels and interfaces. This is because the construction of an environment for

execution of programs offers many opportunities for error, and it is difficult to be

confident that all have been avoided. Volpano and Smith [1999] have pointed out that

an alternative approach, based on proofs that the behaviour of mobile code is sound,

might offer a better solution.

Information leakage • If the transmission of a message between two processes can be

observed, some information can be gleaned from its mere existence – for example, a

flood of messages to a dealer in a particular stock might indicate a high level of trading

in that stock. There are many more subtle forms of information leakage, some malicious

and others arising from inadvertent error. The potential for leakage arises whenever the

results of a computation can be observed. Work was done on the prevention of this type

of security threat in the 1970s [Denning and Denning 1977]. The approach taken is to

assign security levels to information and channels and to analyze the flow of informationSECTION 11.1 INTRODUCTION 485

into channels with the aim of ensuring that high-level information cannot flow into

lower-level channels. A method for the secure control of information flows was first

described by Bell and LaPadula [1975]. The extension of this approach to distributed

systems with mutual distrust between components is the subject of recent research

[Myers and Liskov 1997].

11.1.2 Securing electronic transactions

Many uses of the Internet in industry, commerce and elsewhere involve transactions that

depend crucially on security. For example:

Email: Although email systems did not originally include support for security, there

are many uses of email in which the contents of messages must be kept secret (for

example, when sending a credit card number) or the contents and sender of a message

must be authenticated (for example when submitting an auction bid by email).

Cryptographic security based on the techniques described in this chapter is now

included in many mail clients.

Purchase of goods and services: Such transactions are now commonplace. Buyers

select goods and pay for them via the Web and the goods are delivered through an

appropriate delivery mechanism. Software and other digital products (such as

recordings and videos) can be delivered by downloading. Tangible goods such as

books, CDs and almost every other type of product are also sold by Internet vendors;

these are supplied via a delivery service.

Banking transactions: Electronic banks now offer users virtually all of the facilities

provided by conventional banks. They can check their balances and statements,

transfer money between accounts, set up regular automatic payments and so on.

Micro-transactions: The Internet lends itself to the supply of small quantities of

information and other services to many customers. Most web pages currently can be

viewed without charge, but the development of the Web as a high-quality publishing

medium surely depends upon the ability of information providers to obtain payments

from consumers of the information. Voice and videoconferencing on the Internet is

currently also free, but it is charged for when a telephone network is also involved.

The price for such services may amount to only a fraction of a cent, and the payment

overheads must be correspondingly low. In general, schemes based on the

involvement of a bank or credit card server for each transaction cannot achieve this.

Transactions such as these can be safely performed only when they are protected by

appropriate security policies and mechanisms. A purchaser must be protected against the

disclosure of credit codes (card numbers) during transmission and against fraudulent

vendors who obtain payment with no intention of supplying the goods. Vendors must

obtain payment before releasing the goods, and for downloadable products they must

ensure that only paying customers obtain the data in a usable form. The required

protection must be achieved at a cost that is reasonable in comparison with the value of

the transaction.486 CHAPTER 11 SECURITY

Sensible security policies for Internet vendors and buyers lead to the following

requirements for securing web purchases:

1. Authenticate the vendor to the buyer, so that the buyer can be confident that they

are in contact with a server operated by the vendor with whom they intended to

deal.

2. Keep the buyer’s credit card number and other payment details from falling into

the hands of any third party and ensure that they are transmitted unaltered from

the buyer to the vendor.

3. If the goods are in a form suitable for downloading, ensure that their content is

delivered to the buyer without alteration and without disclosure to third parties.

The identity of the buyer is not normally required by the vendor (except for the purpose

of delivering the goods, if they are not downloaded). The vendor will wish to check that

the buyer has sufficient funds to pay for the purchase, but this is usually done by

demanding payment from the buyer’s bank before delivering the goods.

The security needs of banking transactions using an open network are similar to

those for purchase transactions, with the buyer as the account holder and the bank as the

vendor, but there there is a fourth requirement as well:

4. Authenticate the identity of the account holder to the bank before giving them

access to their account.

Note that in this situation, it is important for the bank to ensure that the account holder

cannot deny that they participated in a transaction. Non-repudiation is the name given

to this requirement.

In addition to the above requirements, which are dictated by security policies,

there are some system requirements. These arise from the very large scale of the

Internet, which makes it impractical to require buyers to enter into special relationships

with vendors (by registering encryption keys for later use, etc.). It should be possible for

a buyer to complete a secure transaction with a vendor even if there has been no previous

contact between buyer and vendor and without the involvement of a third party.

Techniques such as the use of ‘cookies’ – records of previous transactions stored on the

user’s client host – have obvious security weaknesses; desktop and mobile hosts are

often located in insecure physical environments.

Because of the importance of security for Internet commerce and the rapid growth

in Internet commerce, we have chosen to illustrate the use of cryptographic security

techniques by describing in Section 11.6 the de facto standard security protocol used in

most electronic commerce – Transport Layer Security (TLS). A description of Millicent,

a protocol specifically designed for micro-transactions, can be found at

www.cdk5.net/security.

Internet commerce is an important application of security techniques, but it is

certainly not the only one. It is needed wherever computers are used by individuals or

organizations to store and communicate important information. The use of encrypted

email for private communication between individuals is a case in point that has been the

subject of considerable political discussion. We refer to this debate in Section 11.5.2.SECTION 11.1 INTRODUCTION 487

11.1.3 Designing secure systems

Immense strides have been made in recent years in the development of cryptographic

techniques and their application, yet designing secure systems remains an inherently

difficult task. At the heart of this dilemma is the fact that the designer’s aim is to exclude

all possible attacks and loopholes. The situation is analogous to that of the programmer

whose aim is to exclude all bugs from their program. In neither case is there a concrete

method to ensure this goal is met during the design. One designs to the best available

standards and applies informal analysis and checks. Once a design is complete, formal

validation is an option. Work on the formal validation of security protocols has produced

some important results [Lampson et al. 1992, Schneider 1996, Abadi and Gordon 1999].

A description of one of the first steps in this direction, the BAN logic of authentication

[Burrows et al. 1990], and its application can be found at www.cdk5.net/security.

Security is about avoiding disasters and minimizing mishaps. When designing for

security it is necessary to assume the worst. The box on page 488 shows a set of useful

assumptions and design guidelines. These assumptions underly the thinking behind the

techniques that we describe in this chapter.

To demonstrate the validity of the security mechanisms employed in a system, the

system’s designers must first construct a list of threats – methods by which the security

policies might be violated – and show that each of them is prevented by the mechanisms

employed. This demonstration may take the form of an informal argument or, better, a

logical proof.

No list of threats is likely to be exhaustive, so auditing methods must also be used

in security-sensitive applications to detect violations. These are straightforward to

implement if a secure log of security-sensitive system actions is always recorded with

details of the users performing the actions and their authority.

A security log will contain a sequence of timestamped records of users’ actions.

At a minimum the records will include the identity of a principal, the operation

performed (e.g., delete file, update accounting record), the identity of the object

operated on and a timestamp. Where particular violations are suspected, the records may

be extended to include physical resource utilization (network bandwidth, peripherals),

or the logging process may be targeted at operations on particular objects. Subsequent

analysis may be statistical or search-based. Even when no violations are suspected, the

statistics may be compared over time to help to discover any unusual trends or events.

The design of secure systems is an exercise in balancing costs against the threats.

The range of techniques that can be deployed for protecting processes and securing

interprocess communication are strong enough to withstand almost any attack, but their

use incurs expense and inconvenience:

• A cost (in terms of computational effort and network usage) is incurred for their

use. The costs must be balanced against the threats.

• Inappropriately specified security measures may exclude legitimate users from

performing necessary actions.

Such trade-offs are difficult to identify without compromising security and may seem to

conflict with the advice in the first paragraph of this subsection, but the strength of

security techniques required can be quantified and techniques can be selected based on488 CHAPTER 11 SECURITY

the estimated cost of attacks. The relatively low-cost techniques employed in the

Millicent protocol, described at www.cdk5.net/security provide an example.

As an illustration of the difficulties and mishaps that can arise in the design of

secure systems, we review difficulties that arose with the security design originally

incorporated in the IEEE 802.11 WiFi networking standard in Section 11.6.4.

11.2 Overview of security techniques

The purpose of this section is to introduce the reader to some of the more important

techniques and mechanisms for securing distributed systems and applications. Here we

describe them informally, reserving more rigorous descriptions for Sections 11.3 and

11.4. We use the familiar names for principals introduced in Figure 11.1 and the

notations for encrypted and signed items shown in Figure 11.2.

Worst-case assumptions and design guidelines

Interfaces are exposed: Distributed systems are composed of processes that offer

services or share information. Their communication interfaces are necessarily open

(to allow new clients to access them) – an attacker can send a message to any

interface.

Networks are insecure: For example, message sources can be falsified – messages can

be made to look as though they came from Alice when they were actually sent by

Mallory. Host addresses can be ‘spoofed’ – Mallory can connect to the network with

the same address as Alice and receive copies of messages intended for her.

Limit the lifetime and scope of each secret: When a secret key is first generated we can

be confident that it has not been compromised. The longer we use it and the more

widely it is known, the greater the risk. The use of secrets such as passwords and

shared secret keys should be time-limited, and sharing should be restricted.

Algorithms and program code are available to attackers: The bigger and the more

widely distributed a secret is, the greater the risk of its disclosure. Secret encryption

algorithms are totally inadequate for today’s large-scale network environments. Best

practice is to publish the algorithms used for encryption and authentication, relying

only on the secrecy of cryptographic keys. This helps to ensure that the algorithms

are strong by throwing them open to scrutiny by third parties.

Attackers may have access to large resources: The cost of computing power is rapidly

decreasing. We should assume that attackers will have access to the largest and most

powerful computers projected in the lifetime of a system, then add a few orders of

magnitude to allow for unexpected developments.

Minimize the trusted base: The portions of a system that are responsible for the

implementation of its security, and all the hardware and software components upon

which they rely, have to be trusted – this is often referred to as the trusted computing

base. Any defect or programming error in this trusted base can produce security

weaknesses, so we should aim to minimize its size. For example, application

programs should not be trusted to protect data from their users.SECTION 11.2 OVERVIEW OF SECURITY TECHNIQUES 489

11.2.1 Cryptography

Encryption is the process of encoding a message in such a way as to hide its contents.

Modern cryptography includes several secure algorithms for encrypting and decrypting

messages. They are all based on the use of secrets called keys. A cryptographic key is a

parameter used in an encryption algorithm in such a way that the encryption cannot be

reversed without knowledge of the key.

There are two main classes of encryption algorithm in general use. The first uses

shared secret keys – the sender and the recipient must share a knowledge of the key and

it must not be revealed to anyone else. The second class of encryption algorithms uses

public/private key pairs. Here the sender of a message uses a public key – one that has

already been published by the recipient – to encrypt the message. The recipient uses a

corresponding private key to decrypt the message. Although many principals may

examine the public key, only the recipient can decrypt the message, because they have

the private key.

Both classes of encryption algorithm are extremely useful and are used widely in

the construction of secure distributed systems. Public-key encryption algorithms

typically require 100 to 1000 times as much processing power as secret-key algorithms,

but there are situations where their convenience outweighs this disadvantage.

11.2.2 Uses of cryptography

Cryptography plays three major roles in the implementation of secure systems. We

introduce them here in outline by means of some simple scenarios. In later sections of

this chapter, we describe these and other protocols in greater detail, addressing some

unresolved problems that are merely highlighted here.

In all of our scenarios below, we can assume that Alice, Bob and any other

participants have already agreed about the encryption algorithms that they wish to use

and have implementations of them. We also assume that any secret keys or private keys

that they hold can be stored securely to prevent attackers obtaining them.

Secrecy and integrity • Cryptography is used to maintain the secrecy and integrity of

information whenever it is exposed to potential attacks – for example, during

transmission across networks that are vulnerable to eavesdropping and message

tampering. This use of cryptography corresponds to its traditional role in military and

Figure 11.2 Cryptography notations

KA Alice’s secret key

KB Bob’s secret key

KAB Secret key shared between Alice and Bob

KApriv Alice’s private key (known only to Alice)

KApub Alice’s public key (published by Alice for all to read)

{M}K Message M encrypted with key K

[M]K Message M signed with key K490 CHAPTER 11 SECURITY

intelligence activities. It exploits the fact that a message that is encrypted with a

particular encryption key can only be decrypted by a recipient who knows the

corresponding decryption key. Thus it maintains the secrecy of the encrypted message

as long as the decryption key is not compromised (disclosed to non-participants in the

communication) and provided that the encryption algorithm is strong enough to defeat

any possible attempts to crack it. Encryption also maintains the integrity of the

encrypted information, provided that some redundant information such as a checksum

is included and checked.

Scenario 1. Secret communication with a shared secret key: Alice wishes to send some information secretly to Bob. Alice and Bob share a secret key KAB.

1. Alice uses KAB and an agreed encryption function E(KAB, M) to encrypt and send

any number of messages {Mi}KAB to Bob. (Alice can go on using KAB as long as

it is safe to assume that KAB has not been compromised.)

2. Bob decrypts the encrypted messages using the corresponding decryption function

D(KAB, M).

Bob can now read the original message M. If the decrypted message makes sense, or

better, if it includes some value agreed between Alice and Bob (such as a checksum of

the message) then Bob knows that the message is from Alice and that it hasn’t been

tampered with. But there are still some problems:

Problem 1: How can Alice send a shared key KAB to Bob securely?

Problem 2: How does Bob know that any {Mi} isn’t a copy of an earlier encrypted

message from Alice that was captured by Mallory and replayed later? Mallory

needn’t have the key KAB to carry out this attack – he can simply copy the bit pattern

that represents the message and send it to Bob later. For example, if the message is a

request to pay some money to someone, Mallory might trick Bob into paying twice.

We show how these problems can be resolved later in this chapter.

Authentication • Cryptography is used in support of mechanisms for authenticating

communication between pairs of principals. A principal who decrypts a message

successfully using a particular key can assume that the message is authentic if it contains

a correct checksum or (if the block-chaining mode of encryption, described in Section

11.3, is used) some other expected value. They can infer that the sender of the message

possessed the corresponding encryption key and hence deduce the identity of the sender

if the key is known only to two parties. Thus if keys are held in private, a successful

decryption authenticates the decrypted message as coming from a particular sender.

Scenario 2. Authenticated communication with a server: Alice wishes to access files held

by Bob, a file server on the local network of the organization where she works. Sara is

an authentication server that is securely managed. Sara issues users with passwords and

holds current secret keys for all of the principals in the system it serves (generated by

applying some transformation to the user’s password). For example, it knows Alice’s

key KA and Bob’s KB. In our scenario we refer to a ticket. A ticket is an encrypted item

issued by an authentication server, containing the identity of the principal to whom it is

issued and a shared key that has been generated for the current communication session.SECTION 11.2 OVERVIEW OF SECURITY TECHNIQUES 491

1. Alice sends an (unencrypted) message to Sara stating her identity and requesting

a ticket for access to Bob.

2. Sara sends a response to Alice encrypted in KA consisting of a ticket (to be sent to

Bob with each request for file access) encrypted in KB and a new secret key KAB

for use when communicating with Bob. So the response that Alice receives looks

like this: {{Ticket}KB, KAB}KA.

3. Alice decrypts the response using KA (which she generates from her password

using the same transformation; the password is not transmitted over the network,

and once it has been used it is deleted from local storage to avoid compromising

it). If Alice has the correct password-derived key KA, she obtains a valid ticket for

using Bob’s service and a new encryption key for use in communicating with Bob.

Alice can’t decrypt or tamper with the ticket, because it is encrypted in KB. If the

recipient isn’t Alice then they won’t know Alice’s password, so they won’t be able

to decrypt the message.

4. Alice sends the ticket to Bob together with her identity and a request R to access

a file: {Ticket}KB, Alice, R.

5. The ticket, originally created by Sara, is actually: {KAB, Alice}KB. Bob decrypts

the ticket using his key KB. So Bob gets the authentic identity of Alice (based on

the knowledge shared between Alice and Sara of Alice’s password) and a new

shared secret key KAB for use when interacting with Alice. (This is called a session

key because it can safely be used by Alice and Bob for a sequence of interactions.)

This scenario is a simplified version of the authentication protocol originally developed

by Roger Needham and Michael Schroeder [1978] and subsequently used in the

Kerberos system developed and used at MIT [Steiner et al. 1988], which is described in

Section 11.6.2. In our simplified description of their protocol above there is no

protection against the replay of old authentication messages. This and some other

weaknesses are dealt with in our description of the full Needham–Schroeder protocol in

Section 11.6.1.

The authentication protocol we have described depends upon prior knowledge by

the authentication server Sara of Alice’s and Bob’s keys, KA and KB. This is feasible in

a single organization where Sara runs on a physically secure computer and is managed

by a trusted principal who generates initial values of the keys and transmits them to users

by a separate secure channel. But it isn’t appropriate for electronic commerce or other

wide area applications, where the use of a separate channel is extremely inconvenient

and the requirement for a trusted third party is unrealistic. Public-key cryptography

rescues us from this dilemma.

The usefulness of challenges: An important aspect of Needham and Schroeder’s 1978

breakthrough was the realization that a user’s password does not have to be submitted

to an authentication service (and hence exposed in the network) each time it is

authenticated. Instead, they introduced the concept of a cryptographic challenge. This

can be seen in step 2 of our scenario above, where the server, Sara, issues a ticket to

Alice encrypted in Alice’s secret key, KA. This constitutes a challenge because Alice

cannot make use of the ticket unless she can decrypt it, and she can only decrypt it if she492 CHAPTER 11 SECURITY

can determine KA, which is derived from Alice’s password. An imposter claiming to be

Alice would be defeated at this point.

Scenario 3. Authenticated communication with public keys: Assuming that Bob has generated a public/private key pair, the following dialogue enables Bob and Alice to establish

a shared secret key, KAB:

1. Alice accesses a key distribution service to obtain a public-key certificate giving

Bob’s public key. It’s called a certificate because it is signed by a trusted authority

– a person or organization that is widely known to be reliable. After checking the

signature, she reads Bob’s public key, KBpub, from the certificate. (We discuss the

construction and use of public-key certificates in Section 11.2.3.)

2. Alice creates a new shared key, KAB, and encrypts it using KBpub with a publickey algorithm. She sends the result to Bob, along with a name that uniquely

identifies a public/private key pair (since Bob may have several of them) – that is,

Alice sends keyname,{KAB}KBpub.

3. Bob selects the corresponding private key, KBpriv, from his private key store and

uses it to decrypt KAB. Note that Alice’s message to Bob might have been

corrupted or tampered with in transit. The consequence would simply be that Bob

and Alice don’t share the same key KAB. If this is a problem, it can be

circumvented by adding an agreed value or string to the message, such as Bob’s

and Alice’s names or email addresses, which Bob can check after decrypting.

The above scenario illustrates the use of public-key cryptography to distribute a shared

secret key. This technique is known as a hybrid cryptographic protocol and is very

widely used, since it exploits useful features of both public-key and secret-key

encryption algorithms.

Problem: This key exchange is vulnerable to man-in-the-middle attacks. Mallory

may intercept Alice’s initial request to the key distribution service for Bob’s publickey certificate and send a response containing his own public key. He can then

intercept all the subsequent messages. In our description above, we guard against this

attack by requiring Bob’s certificate to be signed by a well-known authority. To

protect against this attack, Alice must ensure that Bob’s public-key certificate is

signed with a public key (as described below) that she has received in a totally secure

manner.

Digital signatures • Cryptography is used to implement a mechanism known as a

digital signature. This emulates the role of a conventional signature, verifying to a third

party that a message or a document is an unaltered copy of one produced by the signer.

Digital signature techniques are based upon an irreversible binding to the message

or document of a secret known only to the signer. This can be achieved by encrypting

the message – or better, a compressed form of the message called a digest – using a key

that is known only to the signer. A digest is a fixed-length value computed by applying

a secure digest function. A secure digest function is similar to a checksum function, but

it is very unlikely to produce a similar digest value for two different messages. The

resulting encrypted digest acts as a signature that accompanies the message. Public-key

cryptography is generally used for this: the originator generates a signature with their

private key, and the signature can be decrypted by any recipient using the correspondingSECTION 11.2 OVERVIEW OF SECURITY TECHNIQUES 493

public key. There is an additional requirement: the verifier should be sure that the public

key really is that of the principal claiming to be the signer – this is dealt with by the use

of public-key certificates, described in Section 11.2.3.

Scenario 4. Digital signatures with a secure digest function: Alice wants to sign a document

M so that any subsequent recipient can verify that she is the originator of it. Thus when

Bob later accesses the signed document after receiving it by any route and from any

source (for example, it could be sent in a message or it could be retrieved from a

database), he can verify that Alice is the originator.

1. Alice computes a fixed-length digest of the document, Digest(M).

2. Alice encrypts the digest in her private key, appends it to M and makes the result,

M, {Digest(M)}KApriv, available to the intended users.

3. Bob obtains the signed document, extracts M and computes Digest(M).

4. Bob decrypts {Digest(M)}KApriv using Alice’s public key, KApub, and compares

the result with his calculated Digest(M). If they match, the signature is valid.

11.2.3 Certificates

A digital certificate is a document containing a statement (usually short) signed by a

principal. We illustrate the concept with a scenario.

Scenario 5. The use of certificates: Bob is a bank. When his customers establish contact

with him they need to be sure that they are talking to Bob the bank, even if they have

never contacted him before. Bob needs to authenticate his customers before he gives

them access to their accounts.

For example, Alice might find it useful to obtain a certificate from her bank stating

her bank account number (Figure 11.3). Alice could use this certificate when shopping

to certify that she has an account with Bob’s Bank. The certificate is signed using Bob’s

private key, KBpriv. A vendor, Carol, can accept such a certificate for charging items to

Alice’s account provided that she can validate the signature in field 5. To do so, Carol

needs to have Bob’s public key and she needs to be sure that it is authentic to guard

against the possibility that Alice might sign a false certificate associating her name with

someone else’s account. To carry out this attack, Alice would simply generate a new key

pair, KB'pub, KB'priv, and use them to generate a forged certificate purporting to come

from Bob’s Bank.

Figure 11.3 Alice’s bank account certificate

1. Certificate type: Account number

2. Name: Alice

3. Account: 6262626

4. Certifying authority: Bob’s Bank

5. Signature: {Digest(field 2 + field 3)}KBpriv494 CHAPTER 11 SECURITY

What Carol needs is a certificate stating Bob’s public key, signed by a well-known

and trusted authority. Let us assume that Fred represents the Bankers Federation, one of

whose roles is to certify the public keys of banks. Fred could issue a public-key

certificate for Bob (Figure 11.4).

Of course, this certificate depends upon the authenticity of Fred’s public key,

KFpub, so we have a recursive problem of authenticity – Carol can only rely on this

certificate if she can be sure she knows Fred’s authentic public key, KFpub. We can break

this recursion by ensuring that Carol obtains KFpub by some means in which she can

have confidence – she might be handed it by a representative of Fred or she might

receive a signed copy of it from someone she knows and trusts who says they got it

directly from Fred. Our example illustrates a certification chain – one with two links, in

this case.

We have already alluded to one of the problems arising with certificates – the

difficulty of choosing a trusted authority from which a chain of authentications can start.

Trust is seldom absolute, so the choice of an authority must depend upon the purpose to

which the certificate is to be put. Other problems arise over the risk of private keys being

compromised (disclosed) and the permissible length of a certification chain – the longer

the chain, the greater the risk of a weak link.

Provided that care is taken to address these issues, chains of certificates are an

important cornerstone for electronic commerce and other kinds of real-world

transaction. They help to address the problem of scale: there are six billion people in the

world, so how can we construct an electronic environment in which we can establish the

credentials of any of them?

Certificates can be used to establish the authenticity of many types of statement.

For example, the members of a group or association might wish to maintain an email list

that is open only to members of the group. A good way to do this would be for the

membership manager (Bob) to issue a membership certificate (S,Bob,{Digest(S)}KBpriv)

to each member, where S is a statement of the form Alice is a member of the Friendly

Society and KBpriv is Bob’s private key. A member applying to join the Friendly Society

email list would have to supply a copy of this certificate to the list management system,

which checks the certificate before allowing the member to join the list.

To make certificates useful, two things are needed:

• a standard format and representation for them so that certificate issuers and

certificate users can successfully construct and interpret them;

Figure 11.4 Public-key certificate for Bob’s Bank

1. Certificate type: Public key

2. Name: Bob’s Bank

3. Public key: KBpub

4. Certifying authority: Fred – The Bankers Federation

5. Signature: {Digest(field 2 + field 3)}KFprivSECTION 11.2 OVERVIEW OF SECURITY TECHNIQUES 495

• agreement on the manner in which chains of certificates are constructed, and in

particular the notion of a trusted authority.

We return to these requirements in Section 11.4.4.

There is sometimes a need to revoke a certificate – for example, Alice might

discontinue her membership of the Friendly Society, but she and others would probably

continue to hold stored copies of her membership certificate. It would be expensive, if

not impossible, to track down and delete all such certificates, and it is not easy to

invalidate a certificate – it would be necessary to notify all possible recipients of the

revocation. The usual solution to this problem is to include an expiry date in the

certificate. Anyone receiving an expired certificate should reject it, and the subject of

the certificate must request its renewal. If a more rapid revocation is required, then one

of the more cumbersome mechanisms mentioned above must be resorted to.

11.2.4 Access control

Here we outline the concepts on which the control of access to resources is based in

distributed systems and the techniques by which it is implemented. The conceptual basis

for protection and access control was very clearly set out in a classic paper by Lampson

[1971], and details of non-distributed implementations can be found in many books on

operating systems (see e.g., [Stallings 2008]).

Historically, the protection of resources in distributed systems has been largely

service-specific. Servers receive request messages of the form <op, principal,

resource>, where op is the requested operation, principal is an identity or a set of

credentials for the principal making the request and resource identifies the resource to

which the operation is to be applied. The server must first authenticate the request

message and the principal’s credentials and then apply access control, refusing any

request for which the requesting principal does not have the necessary access rights to

perform the requested operation on the specified resource.

In object-oriented distributed systems there may be many types of object to which

access control must be applied, and the decisions are often application-specific. For

example, Alice may be allowed only one cash withdrawal from her bank account per

day, while Bob is allowed three. Access control decisions are usually left to the

application-level code, but generic support is provided for much of the machinery that

supports the decisions. This includes the authentication of principals, the signing and

authentication of requests, and the management of credentials and access rights data.

Protection domains • A protection domain is an execution environment shared by a

collection of processes: it contains a set of <resource, rights> pairs, listing the resources

that can be accessed by all processes executing within the domain and specifying the

operations permitted on each resource. A protection domain is usually associated with a

given principal – when a user logs in, their identity is authenticated and a protection

domain is created for the processes that they will run. Conceptually, the domain includes

all of the access rights that the principal possesses, including any rights that they acquire

through membership of various groups. For example, in UNIX, the protection domain

of a process is determined by the user and group identifiers attached to the process at

login time. Rights are specified in terms of allowed operations. For example, a file might

be readable and writable by one process and only readable by another.496 CHAPTER 11 SECURITY

A protection domain is only an abstraction. Two alternative implementations are

commonly used in distributed systems: capabilities and access control lists.

Capabilities: A set of capabilities is held by each process according to the domain in

which it is located. A capability is a binary value that acts as an access key, allowing the

holder access to certain operations on a specified resource. For use in distributed

systems, where capabilities must be unforgeable, they take a form such as:

Services only supply capabilities to clients when they have authenticated them as

belonging to the claimed protection domain. The list of operations in the capability is a

subset of the operations defined for the target resource and is often encoded as a bit map.

Different capabilities are used for different combinations of access rights to the same

resource.

When capabilities are used, client requests are of the form <op, userid,

capability>. That is, they include a capability for the resource to be accessed instead of

a simple identifier, giving the server immediate proof that the client is authorized to

access the resource identified by the capability with the operations specified by the

capability. An access-control check on a request that is accompanied by a capability

involves only the validation of the capability and a check that the requested operation is

in the set permitted by the capability. This feature is the major advantage of capabilities

– they constitute a self-contained access key, just as a physical key to a door lock is an

access key to the building that the lock protects.

Capabilities share two drawbacks of keys to a physical lock:

Key theft: Anyone who holds the key to a building can use it to gain access, whether

or not they are an authorized holder of the key – they may have stolen the key or

obtained it in some fraudulent manner.

The revocation problem: The entitlement to hold a key changes with time. For

example, the holder may cease to be an employee of the owner of the building, but

they might retain the key, or a copy of it, and use it in an unauthorized manner.

The only available solutions to these problems for physical keys are (a) to put the illicit

key holder in jail – not always feasible on a timescale that will prevent them doing

damage – or (b) to change the lock and reissue keys to all key holders – a clumsy and

expensive operation.

The analogous problems for capabilities are clear:

• Capabilities may, through carelessness or as a result of an eavesdropping attack,

fall into the hands of principals other than those to whom they were issued. If this

happens, servers are powerless to prevent them being used illicitly.

• It is difficult to cancel capabilities. The status of the holder may change and their

access rights should change accordingly, but they can still use their capabilities.

Solutions to both of these problems, based on the inclusion of information identifying

the holder and on timeouts plus lists of revoked capabilities, respectively, have been

proposed and developed [Gong 1989, Hayton et al. 1998]. Although they add

Resource identifier A unique identifier for the target resource

Operations A list of the operations permitted on the resource

Authentication code A digital signature making the capability unforgeableSECTION 11.2 OVERVIEW OF SECURITY TECHNIQUES 497

complexity to an otherwise simple concept, capabilities remain an important technique

– for example, they can be used in conjunction with access control lists to optimize

access control on repeated access to the same resource, and they provide the neatest

mechanism for the implementation of delegation (see Section 11.2.5).

It is interesting to note the similarity between capabilities and certificates.

Consider Alice’s certificate of ownership of her bank account introduced in Section

11.2.3. It differs from capabilities as described here only in that there is no list of

permitted operations and that the issuer is identified. Certificates and capabilities may

be interchangeable concepts in some circumstances. Alice’s certificate might be

regarded as an access key allowing her to perform all the operations permitted to account

holders on her bank account, provided her identity can be proven.

Access control lists: A list is stored with each resource, containing an entry of the form

<domain, operations> for each domain that has access to the resource and giving the

operations permitted to the domain. A domain may be specified by an identifier for a

principal or it may be an expression that can be used to determine a principal’s

membership of the domain. For example, the owner of this file is an expression that can

be evaluated by comparing the requesting principal’s identity with the owner’s identity

stored with a file.

This is the scheme adopted in most file systems, including UNIX and

Windows NT, where a set of access permission bits is associated with each file, and the

domains to which the permissions are granted are defined by reference to the ownership

information stored with each file.

Requests to servers are of the form <op, principal, resource>. For each request,

the server authenticates the principal and checks to see that the requested operation is

included in the principal’s entry in the access control list of the relevant resource.

Implementation • Digital signatures, credentials and public-key certificates provide the

cryptographic basis for secure access control. Secure channels offer performance

benefits, enabling multiple requests to be handled without a need for repeated checking

of principals and credentials [Wobber et al. 1994].

Both CORBA and Java offer Security APIs. Support for access control is one of

their major purposes. Java provides support for distributed objects to manage their own

access control with Principal, Signer and ACL classes and default methods for

authentication and support for certificates, signature validation and access-control

checks. Secret-key and public-key cryptography are also supported. Farley [1998]

provides a good introduction to these features of Java. The protection of Java programs

that include mobile code is based upon the protection domain concept – local code and

downloaded code are provided with different protection domains in which to execute.

There can be a protection domain for each download source, with access rights for

different sets of local resources depending upon the level of trust that is placed in the

downloaded code.

Corba offers a Security Service specification [Blakley 1999, OMG 2002b] with a

model for ORBs to provide secure communication, authentication, access control with

credentials, ACLs and auditing; these are described further in Section 8.3.498 CHAPTER 11 SECURITY

11.2.5 Credentials

Credentials are a set of evidence provided by a principal when requesting access to a

resource. In the simplest case, a certificate from a relevant authority stating the

principal’s identity is sufficient, and this would be used to check the principal’s

permissions in an access control list (see Section 11.2.4). This is often all that is required

or provided, but the concept can be generalized to deal with many more subtle

requirements.

It is not convenient to require users to interact with the system and authenticate

themselves each time their authority is required to perform an operation on a protected

resource. Instead, the notion that a credential speaks for a principal is introduced. Thus

a user’s public-key certificate speaks for that user – any process receiving a request

authenticated with the user’s private key can assume that the request was issued by that

user.

The speaks for idea can be carried much further. For example, in a cooperative

task, it might be required that certain sensitive actions should only be performed with

the authority of two members of the team; in that case, the principal requesting the action

would submit their own identifying credential and a backing credential from another

member of the team, together with an indication that they are to be taken together when

checking the credentials.

Similarly, to vote in an election, a vote request would be accompanied by an

elector certificate as well as an identifying certificate. A delegation certificate allows a

principal to act on behalf of another, and so on. In general, an access-control check

involves the evaluation of a logical formula combining the certificates supplied.

Lampson et al. [1992] have developed a comprehensive logic of authentication for use

in evaluating the speaks for authority carried by a set of credentials. Wobber et al. [1994]

describe a system that supports this very general approach. Further work on useful forms

of credential for use in real-world cooperative tasks can be found in Rowley [1998].

Role-based credentials seem particularly useful in the design of practical access

control schemes [Sandhu et al. 1996]. Sets of role-based credentials are defined for

organizations or for cooperative tasks, and application-level access rights are

constructed with reference to them. Roles can then be assigned to specific principals by

the generation of role certificates associating principals with named roles in specific

tasks or organizations [Coulouris et al. 1998].

Delegation • A particularly useful form of credential is one that entitles a principal, or

a process acting for a principal, to perform an action with the authority of another

principal. A need for delegation can arise in any situation where a service needs to

access a protected resource in order to complete an action on behalf of its client.

Consider the example of a print server that accepts requests to print files. It would be

wasteful of resources to copy the file, so the name of the file is passed to the print server

and it is accessed by the print server on behalf of the user making the request. If the file

is read-protected, this does not work unless the print server can acquire temporary rights

to read the file. Delegation is a mechanism designed to solve problems such as this.

Delegation can be achieved using a delegation certificate or a capability. The

certificate is signed by the requesting principal and it authorizes another principal (the

print server in our example) to access a named resource (the file to be printed). In

systems that support them, capabilities can achieve the same result without the need toSECTION 11.2 OVERVIEW OF SECURITY TECHNIQUES 499

identify the principals – a capability to access a resource can be passed in a request to a

server. The capability is an unforgeable, encoded set of rights to access the resource.

When rights are delegated, it is common to restrict them to a subset of the rights

held by the issuing principal, so that the delegated principal cannot misuse them. In our

example, the certificate could be time-limited to reduce the risk of the print server’s code

subsequently being compromised and the file disclosed to third parties. The CORBA

Security Service includes a mechanism for the delegation of rights based on certificates,

with support for the restriction of the rights carried.

11.2.6 Firewalls

Firewalls were introduced and described in Section 3.4.8. They protect intranets,

performing filtering actions on incoming and outgoing communications. Here we

discuss their advantages and drawbacks as security mechanisms.

In an ideal world, communication would always be between mutually trusting

processes and secure channels would always be used. There are many reasons why this

ideal is not attainable, some fixable, but others inherent in the open nature of distributed

systems or resulting from the errors that are present in most software. The ease with

which request messages can be sent to any server, anywhere, and the fact that many

servers are not designed to withstand malicious attacks from hackers or accidental

errors, makes it easy for information that is intended to be confidential to leak out of the

owning organization’s servers. Undesirable items can also penetrate an organization’s

network, allowing worm programs and viruses to enter its computers. See

[web.mit.edu II] for a further critique of firewalls.

Firewalls produce a local communication environment in which all external

communication is intercepted. Messages are forwarded to the intended local recipient

only for communications that are explicitly authorized.

Access to internal networks may be controlled by firewalls, but access to public

services on the Internet is unrestricted because their purpose is to offer services to a wide

range of users. The use of firewalls offers no protection against attacks from inside an

organization, and it is crude in its control of external access. There is a need for finergrained security mechanisms, enabling individual users to share information with

selected others without compromising privacy and integrity. Abadi et al. [1998]

describe an approach to the provision of access to private web data for external users

based on a web tunnel mechanism that can be integrated with a firewall. It offers access

for trusted and authenticated users to internal web servers via a secure proxy based on

the HTTPS (HTTP over TLS) protocol.

Firewalls are not particularly effective against denial-of-service attacks such as

the one based on IP spoofing that was outlined in Section 3.4.2. The problem is that the

flood of messages generated by such attacks overwhelms any single point of defence

such as a firewall. Any remedy for incoming floods of messages must be applied well

upstream of the target. Remedies based on the use of quality of service mechanisms to

restrict the flow of messages from the network to a level that the target can handle seem

the most promising.500 CHAPTER 11 SECURITY

11.3 Cryptographic algorithms

A message is encrypted by the sender applying some rule to transform the plaintext

message (any sequence of bits) to a ciphertext (a different sequence of bits). The

recipient must know the inverse rule in order to transform the ciphertext back into the

original plaintext. Other principals are unable to decipher the message unless they also

know the inverse rule. The encryption transformation is defined with two parts, a

function E and a key K. The resulting encrypted message is written .

The encryption function E defines an algorithm that transforms data items in plaintext

into encrypted data items by combining them with the key and transposing them in a

manner that is heavily dependent on the value of the key. We can think of an encryption

algorithm as the specification of a large family of functions from which a particular

member is selected by any given key. Decryption is carried out using an inverse function

D, which also takes a key as a parameter. For secret-key encryption, the key used for

decryption is the same as that used for encryption:

Because of its symmetrical use of keys, secret-key cryptography is often referred to as

symmetric cryptography, whereas public-key cryptography is referred to as asymmetric

because the keys used for encryption and decryption are different, as we shall see below.

In the next section, we describe several widely used encryption functions of both types.

Symmetric algorithms • If we remove the key parameter from consideration by defining

, then it is a property of strong encryption functions that

is relatively easy to compute, whereas the inverse, , is so hard to

compute that it is not feasible. Such functions are known as one-way functions. The

effectiveness of any method for encrypting information depends upon the use of an

encryption function FK that has this one-way property. It is this that protects against

attacks designed to discover M given .

For well-designed symmetric algorithms such as those described in the next

section, their strength against attempts to discover K given a plaintext M and the

corresponding ciphertext depends on the size of K. This is because the most

effective general form of attack is the crudest, known as a brute-force attack. The bruteforce approach is to run through all possible values of K, computing until the

result matches the value of that is already known. If K has N bits then such an

attack requires iterations on average, and a maximum of iterations, to find K.

Hence the time to crack K is exponential in the number of bits in K.

Asymmetric algorithms • When a public/private key pair is used, one-way functions are

exploited in another way. The feasibility of a public-key scheme was first proposed by

Diffie and Hellman [1976] as a cryptographic method that eliminates the need for trust

between the communicating parties. The basis for all public-key schemes is the

existence of trap-door functions. A trap-door function is a one-way function with a

secret exit – it is easy to compute in one direction but infeasible to compute the inverse

unless a secret is known. It was the possibility of finding such functions and using them

{ } M K

E K M ( ) , = { } M K

D K E K M ( ) , ( ) , = M

FK

( ) [ ] M = E K M ( ) ,

FK

( ) [ ] M FK–1( ) [ ] M

{ } M K

{ } M K

E K M ( ) ,

{ } M K

2N – 1 2NSECTION 11.3 CRYPTOGRAPHIC ALGORITHMS 501

in practical cryptography that Diffie and Hellman first suggested. Since then, several

practical public-key schemes have been proposed and developed. They all depend upon

the use of trap-door functions involving large numbers.

The pair of keys needed for asymmetric algorithms is derived from a common

root. For the RSA algorithm, described in Section 11.3.2, the root is an arbitrarily chosen

pair of very large prime numbers. The derivation of the pair of keys from the root is a

one-way function. In the case of the RSA algorithm, the large primes are multiplied

together – a computation that takes only a few seconds, even for the very large primes

used. The resulting product, N, is of course much larger than the multiplicands. This use

of multiplication is a one-way function in the sense that it is computationally infeasible

to derive the original multiplicands from the product – that is, to factorize the product.

One of the pair of keys is used for encryption. For RSA, the encryption function

obscures the plaintext by treating each block of bits as a binary number and raising it to

the power of the key, modulo N. The resulting number is the corresponding ciphertext

block.

The size of N and at least one of the pair of keys is much larger than the safe key

size for symmetric keys to ensure that N is not factorizable. For this reason, the potential

for brute-force attacks on RSA is small; its resistance to attacks depends on the

infeasibility of factorizing N. We discuss safe sizes for N in Section 11.3.2.

Block ciphers • Most encryption algorithms operate on fixed-size blocks of data; 64 bits

is a popular size for the blocks. A message is subdivided into blocks, the last block is

padded to the standard length if necessary and each block is encrypted independently.

The first block is available for transmission as soon as it has been encrypted.

For a simple block cipher, the value of each block of ciphertext does not depend

upon the preceding blocks. This constitutes a weakness, since an attacker can recognize

repeated patterns and infer their relationship to the plaintext. Nor is the integrity of

messages guaranteed unless a checksum or secure digest mechanism is used. Most block

cipher algorithms employ cipher block chaining (CBC) to overcome these weaknesses.

Cipher block chaining: In cipher block chaining mode, each plaintext block is combined

with the preceding ciphertext block using the exclusive-or operation (XOR) before it is

encrypted (Figure 11.5). On decryption, the block is decrypted and then the preceding

encrypted block (which should have been stored for this purpose) is XOR-ed with it to

obtain the new plaintext block. This works because the XOR operation is its own inverse

– two applications of it produce the original value.

CBC is intended to prevent identical portions of plaintext encrypting to identical

pieces of ciphertext. But there is a weakness at the start of each sequence of blocks – if

n

n+3 n+2

Figure 11.5 Cipher block chaining

n+1 XOR

E(K, M)

n–3 n–2 n–1

plaintext blocks

ciphertext blocks502 CHAPTER 11 SECURITY

we open encrypted connections to two destinations and send the same message, the

encrypted sequences of blocks will be the same, and an eavesdropper might gain some

useful information from this. To prevent this, we need to insert a different piece of

plaintext in front of each message. Such text is called an initialization vector. A

timestamp makes a good initialization vector, forcing each message to start with a

different plaintext block. This, combined with CBC operation, will result in different

ciphertexts even for two identical plaintexts.

The use of CBC mode is restricted to the encryption of data that is transferred

across a reliable connection. Decryption will fail if any blocks of ciphertext are lost,

since the decryption process will be unable to decrypt any further blocks. It is therefore

unsuitable for use in applications such as those described in Chapter 18, in which some

data loss can be tolerated. A stream cipher should be used in such circumstances.

Stream ciphers • For some applications, such as the encryption of telephone

conversations, encryption in blocks is inappropriate because the data streams are

produced in real time in small chunks. Data samples can be as small as 8 bits or even a

single bit, and it would be wasteful to pad each of these to 64 bits before encrypting and

transmitting them. Stream ciphers are encryption algorithms that can perform

encryption incrementally, converting plaintext to ciphertext one bit at a time.

This sounds difficult to achieve, but in fact it is very simple to convert a block

cipher algorithm for use as a stream cipher. The trick is to construct a keystream

generator. A keystream is an arbitrary-length sequence of bits that can be used to

obscure the contents of a data stream by XOR-ing the keystream with the data stream

(Figure 11.6). If the keystream is secure, then so is the resulting encrypted data stream.

The idea is analogous to a technique used in the intelligence community to foil

eavesdroppers, where ‘white noise’ is played to hide the conversation in a room while

still recording the conversation. If the noisy room sound and the white noise are recorded

separately, the conversation can be played back without noise by subtracting the white

noise recording from the noisy room recording.

A keystream generator can be constructed by iterating a mathematical function

over a range of input values to produce a continuous stream of output values. The output

values are then concatenated to make plaintext blocks, and the blocks are encrypted

using a key shared by the sender and the receiver. The keystream can be further

disguised by applying CBC. The resulting encrypted blocks are used as the keystream.

An iteration of almost any function that delivers a range of different non-integer values

will do for the source material, but a random number generator is generally used with a

starting value for the iteration agreed between the sender and receiver. To maintain

quality of service for the data stream, the keystream blocks should be produced a little

Figure 11.6 Stream cipher

XOR

number E(K, M)

generator n+3 n+2 n+1

plaintext

stream

ciphertext

stream

buffer

keystreamSECTION 11.3 CRYPTOGRAPHIC ALGORITHMS 503

ahead of the time at which they will be used, and the process that produces them should

not demand so much processing effort that the data stream is delayed.

Thus in principle, real-time data streams can be encrypted just as securely as

batched data, provided that sufficient processing power is available to encrypt the

keystream in real time. Of course, some devices that could benefit from real-time

encryption, such as mobile phones, are not equipped with very powerful processors, and

in that case it may be necessary to reduce the security of the keystream algorithm.

Design of cryptographic algorithms • There are many well-designed cryptographic

algorithms such that conceals the value of M and makes it

practically impossible to retrieve K more quickly than by brute force. All encryption

algorithms rely on information-preserving manipulations of M using principles based on

information theory [Shannon 1949]. Schneier [1996] describes Shannon’s principles of

confusion and diffusion to conceal the content of a ciphertext block M, combining it with

a key K of sufficient size to render it proof against brute-force attacks.

Confusion: Non-destructive operations such as XOR and circular shifting are used to

combine each block of plaintext with the key, producing a new bit pattern that obscures

the relationship between the blocks in M and {M}K. If the blocks are larger than a few

characters this will defeat analysis based on a knowledge of character frequencies. (The

WWII German Enigma machine used chained single-letter blocks, and was eventually

defeated by statistical analysis.)

Diffusion: There is usually repetition and redundancy in the plaintext. Diffusion

dissipates the regular patterns that result by transposing portions of each plaintext block.

If CBC is used, the redundancy is also distributed throughout a longer text. Stream

ciphers cannot use diffusion since there are no blocks.

In the next two sections, we describe the design of several important practical

algorithms. All of them have been designed in the light of the above principles have been

subject to rigorous analysis and are considered to be secure against all known attacks

with a considerable margin of safety. With the exception of the TEA algorithm, which

is described for illustrative purposes, the algorithms described here are among those

most widely used in applications where strong security is required. In some of them

there remain some minor weaknesses or areas of concern; space does not allow us to

describe all of those concerns here, and the reader is referred to Schneier [1996] for

further information. We summarize and compare the security and performance of the

algorithms in Section 11.5.1.

Readers who do not require an understanding of the operation of cryptographic

algorithms may omit Sections 11.3.1 and 11.3.2.

11.3.1 Secret-key (symmetric) algorithms

Many cryptographic algorithms have been developed and published in recent years.

Schneier [1996] describes more than 25 symmetric algorithms, many of which he

identifies as secure against known attacks. Here we have room to describe only three of

them. We have chosen the first, TEA, for the simplicity of its design and

implementation, and we use it to give a concrete illustration of the nature of such

algorithms. We go on to discuss the DES and IDEA algorithms in less detail. DES was

E K M ( ) , = { } M K504 CHAPTER 11 SECURITY

a US national standard for many years, but it is now largely of historical interest because

its 56-bit keys are too small to resist brute-force attack with modern hardware. IDEA

uses a 128-bit key. It is one of the most effective symmetric block encryption algorithms

and a good all-round choice for bulk encryption.

In 1997, the US National Institute for Standards and Technology (NIST) issued an

invitation for proposals for an algorithm to replace DES as a new US Advanced

Encryption Standard (AES). In October 2000 the winner was selected from 21

algorithms submitted by cryptographers from 11 countries. The winning Rijndael

algorithm was chosen for its combination of strength and efficiency. Further information

on it is given below.

TEA • The design principles for symmetric algorithms outlined above are illustrated

well in the Tiny Encryption Algorithm (TEA) developed at Cambridge University

[Wheeler and Needham 1994]. The encryption function, programmed in C, is given in

its entirety in Figure 11.7.

The TEA algorithm uses rounds of integer addition, XOR (the ^ operator) and

bitwise logical shifts (<< and >>) to achieve diffusion and confusion of the bit patterns

in the plaintext. The plaintext is a 64-bit block represented as two 32-bit integers in the

vector text[]. The key is 128 bits long, represented as four 32-bit integers.

On each of the 32 rounds, the two halves of the text are repeatedly combined with

shifted portions of the key and each other in lines 5 and 6. The use of XOR and shifted

portions of the text provides confusion, and the shifting and swapping of the two

portions of the text provides diffusion. The non-repeating constant delta is combined

with each portion of the text on each cycle to obscure the key in case it might be revealed

by a section of text that does not vary. The decryption function is the inverse of that for

encryption and is given in Figure 11.8.

This short program provides secure and reasonably fast secret-key encryption. It

is somewhat faster than the DES algorithm, and the conciseness of the program lends

itself to optimization and hardware implementation. The 128-bit key is secure against

brute-force attacks. Studies by its authors and others have revealed only two very minor

weaknesses, which the authors addressed in a subsequent note [Wheeler and Needham

1997].

Figure 11.7 TEA encryption function

void encrypt(unsigned long k[], unsigned long text[]) {

unsigned long y = text[0], z = text[1]; 1

unsigned long delta = 0x9e3779b9, sum = 0; int n; 2

for (n= 0; n < 32; n++) { 3

sum += delta; 4

y += ((z << 4) + k[0]) ^ (z+sum) ^ ((z >> 5) + k[1]); 5

z += ((y << 4) + k[2]) ^ (y+sum) ^ ((y >> 5) + k[3]); 6

}

text[0] = y; text[1] = z; 7

}SECTION 11.3 CRYPTOGRAPHIC ALGORITHMS 505

To illustrate its use, Figure 11.9 shows a simple procedure that uses TEA to

encrypt and decrypt a pair of previously opened files (using the C stdio library).

DES • The Data Encryption Standard (DES) [National Bureau of Standards 1977] was

developed by IBM and subsequently adopted as a US national standard for government

and business applications. In this standard, the encryption function maps a 64-bit

plaintext input into a 64-bit encrypted output using a 56-bit key. The algorithm has 16

key-dependent stages known as rounds, in which the data to be encrypted is bit-rotated

by a number of bits determined by the key and three key-independent transpositions.

The algorithm was time-consuming to perform in software on the computers of the

1970s and 1980s, but it was implemented in fast VLSI hardware and can easily be

incorporated into network interface and other communication chips.

Figure 11.8 TEA decryption function

void decrypt(unsigned long k[], unsigned long text[]) {

unsigned long y = text[0], z = text[1];

unsigned long delta = 0x9e3779b9, sum = delta << 5; int n;

for (n= 0; n < 32; n++) {

z -= ((y << 4) + k[2]) ^ (y + sum) ^ ((y >> 5) + k[3]);

y -= ((z << 4) + k[0]) ^ (z + sum) ^ ((z >> 5) + k[1]);

sum -= delta;

}

text[0] = y; text[1] = z;

}

Figure 11.9 TEA in use

void tea(char mode, FILE \*infile, FILE \*outfile, unsigned long k[]) {

/\* mode is 'e' for encrypt, 'd' for decrypt, k[] is the key.\*/

char ch, Text[8]; int i;

while(!feof(infile)) {

i = fread(Text, 1, 8, infile); /\* read 8 bytes from infile into Text \*/

if (i <= 0) break;

while (i < 8) { Text[i++] = ' ';} /\* pad last block with spaces \*/

switch (mode) {

case 'e':

encrypt(k, (unsigned long\*) Text); break;

case 'd':

decrypt(k, (unsigned long\*) Text); break;

}

fwrite(Text, 1, 8, outfile); /\* write 8 bytes from Text to outfile \*/

}

}506 CHAPTER 11 SECURITY

In June 1997, it was successfully cracked in a widely publicized brute-force

attack. The attack was performed in the context of a competition to demonstrate the lack

of security of encryption with keys shorter than 128 bits [www.rsasecurity.com II]. A

consortium of Internet users ran a client application program on a their PCs and other

workstations, whose numbers reached 14,000 during one 24-hour period [Curtin and

Dolske 1998].

The client program was aimed at cracking the particular key used in a known

plaintext/ciphertext sample and then using it to decrypt a secret challenge message. The

clients interacted with a single server that coordinated their work, issuing each client

with ranges of key values to check and receiving progress reports from them. The typical

client computer ran the client program only as a background activity and had a

performance approximately equal to a 200 MHz Pentium processor. The key was

cracked in about 12 weeks, after approximately 25% of the possible 256 or 6 × 1016

values had been checked. In 1998 a machine was developed by the Electronic Frontier

Foundation [EFF 1998] that can successfully crack DES keys in around three days.

Although it is still used in many commercial and other applications, DES in its

basic form should be considered obsolete for the protection of all but low-value

information. A solution that is frequently used is known as triple-DES (or 3DES) [ANSI

1985, Schneier 1996]. This involves applying DES three times with two keys, K1 and K2:

This gives a strength against brute-force attacks equivalent to a key length of 112 bits –

adequate for the foreseeable future – but it has the drawback of poor performance

resulting from the triple application of an algorithm that is already slow by modern

standards.

IDEA • The International Data Encryption Algorithm (IDEA) was developed in the

early 1990s [Lai and Massey 1990, Lai 1992] as a successor to DES. Like TEA, it uses

a 128-bit key to encrypt 64-bit blocks. Its algorithm is based on the algebra of groups

and has eight rounds of XOR, addition modulo 216 and multiplication. For both DES and

IDEA, the same function is used for encryption and decryption: a useful property for

algorithms that are to be implemented in hardware.

The strength of IDEA has been extensively analyzed, and no significant

weaknesses have been found. It performs encryption and decryption at approximately

three times the speed of DES.

RC4 • RC4 is a stream cipher developed by Ronald Rivest [Rivest 1992b]. Keys can be

of any length up to 256 bytes. RC4 is easy to implement [Schneier 1996, pp. 397–8] and

performs encryption and decryption about 10 times as fast as DES. It was therefore

widely adopted in applications including IEEE 802.11 WiFi networks, but a weakness

was subsequently discovered by Fluhrer et al. [2001] that enabled attackers to crack

some keys. This led to a redesign of 802.11 security (see Section 11.6.4 for further

details).

AES • The Rijndael algorithm selected to become the Advanced Encryption Standard

algorithm by NIST was developed by Joan Daemen and Vincent Rijmen [Daemen and

Rijmen 2000, 2002]. The cipher has a variable block length and key length, with

specifications for keys with a length of 128, 192 or 256 bits to encrypt blocks with a

E

3DES( ) K1, , K2 M = EDES( ) K1, DDES( ) K2, EDES( ) K1, MSECTION 11.3 CRYPTOGRAPHIC ALGORITHMS 507

length of 128, 192 or 256 bits. Both block length and key length can be extended by

multiples of 32 bits. The number of rounds in the algorithm varies from 9 to 13

depending on the key and block sizes. Rijndael can be implemented efficiently on a wide

range of processors and in hardware.

11.3.2 Public-key (asymmetric) algorithms

Only a few practical public-key schemes have been developed to date. They depend

upon the use of trap-door functions of large numbers to produce the keys. The keys Ke

and Kd are a pair of very large numbers, and the encryption function performs an

operation, such as exponentiation on M, using one of them. Decryption is a similar

function using the other key. If the exponentiation uses modular arithmetic, it can be

shown that the result is the same as the original value of M; that is:

D(Kd, E(Ke, M)) = M

A principal wishing to participate in secure communication with others makes a pair of

keys, Ke and Kd, and keeps the decryption key Kd a secret. The encryption key Ke can

be made known publicly for use by anyone who wants to communicate. The encryption

key Ke can be seen as a part of the one-way encryption function E, and the decryption

key Kd is the piece of secret knowledge that enables principal p to reverse the

encryption. Any holder of Ke (which is widely available) can encrypt messages {M}Ke,

but only the principal who has the secret Kd can operate the trapdoor.

The use of functions of large numbers leads to large processing costs in computing

the functions E and D. We shall see later that this is a problem that has to be addressed

by the use of public keys only in the initial stages of secure communication sessions. The

RSA algorithm is certainly the most widely known public-key algorithm and we

describe it in some detail here. Another class of algorithms is based on functions derived

from the behaviour of elliptic curves in a plane. These algorithms offer the possibility of

less costly encryption and decryption functions with the same level of security, but their

practical application is less advanced and we deal with them only briefly.

RSA • The Rivest, Shamir and Adelman (RSA) design for a public-key cipher [Rivest

et al. 1978] is based on the use of the product of two very large prime numbers (greater

than 10100), relying on the fact that the determination of the prime factors of such large

numbers is so computationally difficult as to be effectively impossible.

Despite extensive investigations no flaws have been found in it, and it is now very

widely used. An outline of the method follows. To find a key pair <e,d>:

1. Choose two large prime numbers, P and Q (each greater than 10100), and form

N = P × Q

Z = (P–1) × (Q–1)

2. For d, choose any number that is relatively prime with Z (that is, such that d has

no common factors with Z).

We illustrate the computations involved using small integer values for P and Q:

P = 13, Q = 17 → N = 221, Z = 192

d = 5508 CHAPTER 11 SECURITY

3. To find e, solve the equation:

e × d = 1 mod Z

That is, e × d is the smallest element divisible by d in the series Z+1, 2Z+1, 3Z+1, ... .

To encrypt text using the RSA method, the plaintext is divided into equal blocks of

length k bits, where 2k < N (that is, such that the numerical value of a block is always

less than N; in practical applications, k is usually in the range 512 to 1024).

The function for encrypting a single block of plaintext M is:

E'(e,N,M) = Me mod N

The function for decrypting a block of encrypted text c to produce the original plaintext

block is:

D'(d,N,c) = cd mod N

Rivest, Shamir and Adelman proved that E' and D' are mutual inverses (that is,

E'(D'(x)) = D'(E'(x)) = x) for all values of P in the range 0  P  N.

The two parameters e,N can be regarded as a key for the encryption function, and

similarly the parameters d,N represent a key for the decryption function. So we can write

Ke = <e,N> and Kd = <d,N>, and we get the encryption functions E(Ke, M) ={M}K (the

notation here indicating that the encrypted message can be decrypted only by the holder

of the private key Kd) and D(Kd, {M}K) = M.

It is worth noting one potential weakness of all public-key algorithms – because

the public key is available to attackers, they can easily generate encrypted messages.

Thus they can attempt to decrypt an unknown message by exhaustively encrypting

arbitrary bit sequences until a match with the target message is achieved. This attack,

which is known as a chosen plaintext attack, is defeated by ensuring that all messages

are longer than the key length, so that this form of brute-force attack is less feasible than

a direct attack on the key.

An intending recipient of secret information must publish or otherwise distribute

the pair <e,N> while keeping d secret. The publication of <e,N> does not compromise

the secrecy of d, because any attempt to determine d requires knowledge of the original

prime numbers P and Q, and these can only be obtained by the factorization of N.

Factoring of large numbers (we recall that P and Q were chosen to be > 10100, so N >

10200) is extremely time-consuming, even on very high-performance computers. In

1978, Rivest et al. concluded that factoring a number as large as 10200 would take more

than four billion years with the best known algorithm on a computer that performs one

e × d = 1 mod 192 = 1, 193, 385, ...

385 is divisible by d

e = 385/5 = 77

k = 7, since 27 = 128

for a message M, the ciphertext is M77 mod 221SECTION 11.4 DIGITAL SIGNATURES 509

million instructions per second. A similar calculation for today’s computers would

reduce this time to around a million years,

The RSA Corporation has issued a series of challenges to factor numbers of more

than 100 decimal digits [www.rsasecurity.com III]. At the time of writing, numbers of

up to 174 decimal digits (576 binary digits) have been successfully factored, so the use

of the RSA algorithm with 512-bit keys is clearly unacceptably weak for many purposes.

The RSA Corporation (holders of the patents in the RSA algorithm) recommends a key

length of at least 768 bits, or about 230 decimal digits, for long-term (~20 years)

security. Keys as large as 2048 bits are used in some applications.

The above strength calculations assume that the currently known factoring

algorithms are the best available. RSA and other forms of asymmetric cryptography that

use prime number multiplication as their one-way function will be vulnerable if a faster

factorization algorithm is discovered.

Elliptic curve algorithms • A method for generating public/private key pairs based on

the properties of elliptic curves has been developed and tested. Full details can be found

in the book by Menezes devoted to the subject [Menezes 1993]. The keys are derived

from a different branch of mathematics, and unlike RSA their security does not depend

upon the difficulty of factoring large numbers. Shorter keys are secure, and the

processing requirements for encryption and decryption are lower than those for RSA.

Elliptic curve encryption algorithms are likely to be adopted more widely in the future,

especially in systems such as those incorporating mobile devices, which have limited

processing resources. The relevant mathematics involves some quite complex properties

of elliptic curves and is beyond the scope of this book.

11.3.3 Hybrid cryptographic protocols

Public-key cryptography is convenient for electronic commerce because there is no need

for a secure key-distribution mechanism. (There is a need to authenticate public keys,

but this is much less onerous, requiring only a public-key certificate to be sent with the

key.) But the processing costs of public-key cryptography are too high for the encryption

of even the medium-sized messages normally encountered in electronic commerce. The

solution adopted in most large-scale distributed systems is to use a hybrid encryption

scheme in which public-key cryptography is used to authenticate the parties and to

encrypt an exchange of secret keys, which are used for all subsequent communication.

We describe the implementation of a hybrid protocol in the TLS case study in

Section 11.6.3.

11.4 Digital signatures

Strong digital signatures are an essential requirement for secure systems. They are

needed in order to certify certain pieces of information – for example, to provide

trustworthy statements binding users’ identities to their public keys or binding some

access rights or roles to users’ identities.510 CHAPTER 11 SECURITY

The need for signatures in many kinds of business and personal transaction is

beyond dispute. Handwritten signatures have been used as a means of verifying

documents for as long as documents have existed. Handwritten signatures are used to

meet the needs of document recipients to verify that the document is:

Authentic: It convinces the recipient that the signer deliberately signed the document

and it has not been altered by anyone else.

Unforgeable: It provides proof that the signer, and no one else, deliberately signed

the document. The signature cannot be copied and placed on another document.

Non-repudiable: The signer cannot credibly deny that the document was signed by

them.

In reality, none of these desirable properties of signing is entirely achieved by

conventional signatures – forgeries and copies are hard to detect, documents can be

altered after signing and signers are sometimes deceived into signing a document

involuntarily or unwittingly – but we are willing to live with their imperfection because

of the difficulty of cheating and the risk of detection. Like handwritten signatures,

digital signatures depend upon the binding of a unique and secret attribute of the signer

to a document. In the case of handwritten signatures, the secret is the handwriting pattern

of the signer.

The properties of digital documents held in stored files or messages are

completely different from those of paper documents. Digital documents are trivially

easy to generate, copy and alter. Simply appending the identity of the originator,

whether as a text string, a photograph or a handwritten image, has no value for

verification purposes.

What is needed is a means to irrevocably bind a signer’s identity to the entire

sequence of bits representing a document. This should meet the first requirement above,

for authenticity. As with handwritten signatures, though, the date of a document cannot

be guaranteed by a signature. The recipient of a signed document knows only that the

document was signed before they received it.

Regarding non-repudiation, there is a problem that does not arise with handwritten

signatures. What if the signer deliberately reveals their private key and subsequently

denies having signed, saying that there are others who could have done so, since the key

was not private? Some protocols have been developed to address this problem under the

heading of undeniable digital signatures [Schneier 1996], but they add considerably to

the complexity.

A document with a digital signature can be considerably more resistant to forgery

than a handwritten one. But the word ‘original’ has little meaning with reference to

digital documents. As we shall see in our discussion of the needs of electronic

commerce, digital signatures alone cannot, for example, prevent double-spending of

electronic cash – other measures are needed to prevent that. We now describe two

techniques for signing documents digitally, binding a principal’s identity to the

document. Both depend upon the use of cryptography.

Digital signing • An electronic document or message M can be signed by a principal A

by encrypting a copy of M with a key KA and attaching it to a plaintext copy of M and

A’s identifier. The signed document then consists of: M, A, [M]KA. The signature can beSECTION 11.4 DIGITAL SIGNATURES 511

verified by a principal that subsequently receives the document to check that it was

originated by A and that its contents, M, have not subsequently been altered.

If a secret key is used to encrypt the document, only principals that share the secret

can verify the signature. But if public-key cryptography is used, then the signer uses

their private key and anyone who has the corresponding public key can verify the

signature. This is a better analogue for conventional signatures and meets a wider range

of user needs. The verification of signatures proceeds differently depending on whether

secret-key or public-key cryptography is used to produce the signature. We describe the

two cases in Sections 11.4.1 and 11.4.2.

Digest functions • Digest functions are also called secure hash functions and denoted

H(M). They must be carefully designed to ensure that H(M) is different from H(M') for

all likely pairs of messages M and M'. If there are any pairs of different messages M and

M' such that H(M) = H(M'), then a duplicitous principal could send a signed copy of M,

but when confronted with it claim that M' was originally sent and that it must have been

altered in transit. We discuss some secure hash functions in Section 11.4.3.

11.4.1 Digital signatures with public keys

Public-key cryptography is particularly well adapted for the generation of digital

signatures because it is relatively simple and does not require any communication

between the recipient of a signed document and the signer or any third party.

The method for A to sign a message M and B to verify it is as follows (and is

illustrated graphically in Figure 11.10):

1. A generates a key pair Kpub and Kpriv and publishes the public key Kpub by

placing it in a well-known location.

{h }Kpri

M

Signing

Verifying

E(Kpri, h)

128 bits

H(M) h

M

H(doc) h

{h}Kpri D(Kpub, {h }) h'

h = h'?

Figure 11.10 Digital signatures with public keys

M

signed doc512 CHAPTER 11 SECURITY

2. A computes the digest of M, H(M) using an agreed secure hash function H and

encrypts it using the private key Kpriv to produce the signature S = {H(M)}Kpriv.

3. A sends the signed message [M]K = M,S to B.

4. B decrypts S using Kpub and computes the digest of M, H(M). If they match, the

signature is valid.

The RSA algorithm is quite suitable for use in constructing digital signatures. Note that

the private key of the signer is used to encrypt the signature, in contrast to the use of the

recipient’s public key for encryption when the aim is to transmit information in secrecy.

The explanation for this difference is straightforward – a signature must be created using

a secret known only to the signer and it should be accessible to all for verification.

11.4.2 Digital signatures with secret keys – MACs

There is no technical reason why a secret-key encryption algorithm should not be used

to encrypt a digital signature, but in order to verify such signatures the key must be

disclosed, and this causes some problems:

• The signer must arrange for the verifier to receive the secret key used for signing

securely.

• It may be necessary to verify a signature in several contexts and at different times

– at the time of signing, the signer may not know the identities of the verifiers. To

resolve this, verification could be delegated to a trusted third party who holds

secret keys for all signers, but this adds complexity to the security model and

requires secure communication with the trusted third party.

• The disclosure of a secret key used for signing is undesirable because it weakens

the security of signatures made with that key – a signature could be forged by a

holder of the key who is not the owner of it.

For all these reasons, the public-key method for generating and verifying signatures

offers the most convenient solution in most situations.

An exception arises when a secure channel is used to transmit unencrypted

messages but there is a need to verify the authenticity of the messages. Since a secure

channel provides secure communication between a pair of processes, a shared secret key

can be established using the hybrid method outlined in Section 11.3.3 and used to

produce low-cost signatures. These signatures are called message authentication codes

(MACs) to reflect their more limited purpose – they authenticate communication

between pairs of principals based on a shared secret.

A low-cost signing technique based on shared secret keys that has adequate

security for many purposes is illustrated in Figure 11.11 and outlined below. The

method depends upon the existence of a secure channel through which the shared key

can be distributed:

1. A generates a random key K for signing and distributes it using secure channels to

one or more principals who will need to authenticate messages received from A.

The principals are trusted not to disclose the shared key.SECTION 11.4 DIGITAL SIGNATURES 513

2. For any document M that A wishes to sign, A concatenates M with K, computes

the digest of the result, , and sends the signed document

to anyone wishing to verify the signature. (The digest h is a MAC.)

K will not be compromised by the disclosure of h, since the hash function has

totally obscured its value.

3. The receiver, B, concatenates the secret key K with the received document M and

computes the digest . The signature is verified if .

Although this method suffers from the disadvantages listed above, it has a performance

advantage because it involves no encryption. (Secure hashing is typically about 3–10

times faster than symmetric encryption; see Section 11.5.1.) The TLS secure channel

protocol described in Section 11.6.3 supports the use of a wide variety of MACs,

including the scheme described here. The method is also used in the Millicent electronic

cash protocol described at www.cdk5.net/security, where it is important to keep the

processing cost low for low-value transactions.

11.4.3 Secure digest functions

There are many ways to produce a fixed-length bit pattern that characterizes an

arbitrary-length message or document. Perhaps the simplest is to use the XOR operation

iteratively to combine fixed-length pieces of the source document. Such a function is

often used in communication protocols to produce a short fixed-length hash to

characterize a message for error-detection purposes, but it is inadequate as the basis for

Figure 11.11 Low-cost signatures with a shared secret key

M

Signing

Verifying

H(M+K) h

H(M+K) h'

h

h = h'?

K

M

signed doc

M K

h H M K = ( ) +

[ ] M K = M h ,

h' = H M K ( ) + h h = '514 CHAPTER 11 SECURITY

a digital signature scheme. A secure digest function h = H(M) should have the following

properties:

1. Given M, it is easy to compute h.

2. Given h, it is hard to compute M.

3. Given M, it is hard to find another message M', such that H(M) = H(M').

Such functions are also called one-way hash functions. The reason for this name is selfevident based on the first two properties. Property 3 demands an additional feature: even

though we know that the result of a hash function cannot be unique (because the digest

is an information-reducing transformation), we need to be sure that an attacker, given a

message M that produces a hash h, cannot discover another message M' that also

produces h. If an attacker could do this, then they could forge a signed document M'

without knowledge of the signing key by copying the signature from the signed

document M and appending it to M'.

Admittedly, the set of messages that hash to the same value is restricted and the

attacker would have difficulty in producing a meaningful forgery, but with patience it

could be done, so it must be guarded against. The feasibility of doing so is considerably

enhanced in the case of a so-called birthday attack:

1. Alice prepares two versions, M and M', of a contract for Bob. M is favourable to

Bob and M' is not.

2. Alice makes several subtly different versions of both M and M' that are visually

indistinguishable from each other by methods such as adding spaces at the ends of

lines. She compares the hashes of all the Ms with all the M's. If she finds two that

are the same, she can proceed to the next step; if not, she goes on producing

visually indistinguishable versions of the two documents until she gets a match.

3. When she has a pair of documents M and M' that hash to the same value, she gives

the favourable document M to Bob for him to sign with a digital signature using

his private key. When he returns it, she substitutes the matching unfavourable

version M', retaining the signature from M.

If our hash values are 64 bits long, we require only 232 versions of M and M' on average.

This is too small for comfort. We need to make our hash values at least 128 bits long to

guard against this type of attack.

The attack relies on a statistical paradox known as the birthday paradox – the

probability of finding a matching pair in a given set is far greater than that for finding a

match for a given individual. Stallings [2005] gives the statistical derivation for the

probability that there will be two people with the same birthday in a set of n people. The

result is that for a set of only 23 people the chances are even, whereas we require a set

of 253 people for an even chance that there will be one with a birthday on a given day.

To satisfy the properties listed above, a secure digest function needs to be

carefully designed. The bit-level operations used and their sequencing are similar to

those found in symmetric cryptography, but in this case the operations need not be

information-preserving, since the function is definitely not intended to be reversible. So

a secure digest function can make use of the full range of arithmetic and bit-wise logical

operations. The length of the source text is usually included in the digested data.SECTION 11.4 DIGITAL SIGNATURES 515

Two widely used digest functions for practical applications are the MD5

algorithm (so called because it is the fifth in a sequence of message digest algorithms

developed by Ron Rivest) and SHA-1 (the Secure Hash Algorithm), which has been

adopted for standardization by the US National Institute for Standards and Technology

(NIST). Both have been carefully tested and analyzed and can be considered adequately

secure for the foreseeable future, while their implementations are reasonably efficient.

We describe them briefly here. Schneier [1996] and Mitchell et al. [1992] survey digital

signature techniques and message digest functions in depth.

MD5 • The MD5 algorithm [Rivest 1992a] uses four rounds, each applying one of four

nonlinear functions to each of 16 32-bit segments of a 512-bit block of source text. The

result is a 128-bit digest. MD5 is one of the most efficient algorithms currently available.

SHA-1 • SHA-1 [NIST 2002] is an algorithm that produces a 160-bit digest. It is based

on Rivest’s MD4 algorithm (which is similar to MD5), with some additional operations.

It is substantially slower than MD5, but the 160-bit digest does offer greater security

against brute-force and birthday-style attacks. SHA algorithms that deliver longer

digests (224, 256 and 512 bits) are also included in the standard [NIST 2002]. Of course,

their additional length implies additional costs for the generation, storage and

communication of digital signatures and MACs, but following the publication of attacks

on SHA-1’s predecessors, which suggest that SHA-1 is vulnerable [Randall and Szydlo

2004], NIST announced that is to be superseded by the longer SHA digest versions in

US government software by 2010 [NIST 2004].

Using an encryption algorithm to make a digest • It is possible to use a symmetric

encryption algorithm such as those detailed in Section 11.3.1 to produce a secure digest.

In this case, the key should be published so that the digest algorithm can be applied by

anyone wishing to verify a digital signature. The encryption algorithm is used in CBC

mode, and the digest is the result of combining the penultimate CBC value with the final

encrypted block.

11.4.4 Certificate standards and certificate authorities

X.509 is the most widely used standard format for certificates [CCITT 1988b]. Although

the X.509 certificate format is a part of the X.500 standard for the construction of global

directories of names and attributes [CCITT 1988a], it is commonly used in

cryptographic work as a format definition for freestanding certificates. We describe the

X.500 naming standard in Chapter 13.

Figure 11.12 X509 Certificate format

Subject Distinguished Name, Public Key

Issuer Distinguished Name, Signature

Period of validity Not Before Date, Not After Date

Administrative information Version, Serial Number

Extended information516 CHAPTER 11 SECURITY

The structure and content of an X.509 certificate are illustrated in Figure 11.12.

As can be seen, it binds a public key to a named entity called a subject. The binding is

in the signature, which is issued by another named entity called the issuer. The

certificate has a period of validity, which is defined by a pair of dates. The

<Distinguished Name> entries are intended to be the name of a person, organization or

other entity together with sufficient contextual information to render it unique. In a full

X.500 implementation this contextual information would be drawn from a directory

hierarchy in which the named entity appears, but in the absence of global X.500

implementations it can only be a descriptive string.

This format is included in the TLS protocol for electronic commerce and is widely

used in practice to authenticate the public keys of services and their clients. Certain wellknown companies and organizations have established themselves to act as certificate

authorities (for example, Verisign [www.verisign.com] and CREN [www.cren.net]),

and other companies and individuals can obtain X.509 public-key certificates from them

by submitting satisfactory evidence of their identity. This leads to a two-step verification

procedure for any X.509 certificate:

1. Obtain the public-key certificate of the issuer (a certification authority) from a

reliable source.

2. Validate the signature.

The SPKI approach • The X.509 approach is based on the global uniqueness of

distinguished names. It has been pointed out that this is an impractical goal that does not

reflect the reality of current legal and commercial practice [Ellison 1996], in which the

identities of individuals are not assumed to be unique but are made unique by reference

to other people and organizations. This can be seen in the use of a driving licence or a

letter from a bank to authenticate an individual’s name and address (a name alone is

unlikely to be unique among the world’s population). This leads to longer verification

chains, because there are many possible issuers of public-key certificates, and their

signatures must be validated through a chain of verification that leads back to someone

known and trusted by the principal performing the verification. But the resulting

verification is likely to be more convincing, and many of the steps in such a chain can

be cached to shorten the process on future occasions.

The arguments above are the basis for the recently developed Simple Public-Key

Infrastructure (SPKI) proposals (see RFC 2693 [Ellison et al. 1999]). This is a scheme

for the creation and management of sets of public certificates. It enables chains of

certificates to be processed using logical inference to produce derived certificates. For

example, ‘Bob believes that Alice’s public key is KApub’ and ‘Carol trusts Bob on

Alice’s keys’ implies ‘Carol believes that Alice’s public key is KApub’.

11.5 Cryptography pragmatics

In Section 11.5.1, we compare the performance of the encryption and secure hash

algorithms described or mentioned above. We consider encryption algorithms alongside

secure hash functions because encryption can also be used as a method for digital

signing.SECTION 11.5 CRYPTOGRAPHY PRAGMATICS 517

In Section 11.5.2, we discuss some non-technical issues surrounding the use of

cryptography. There is not space to do justice to the vast amount of political discussion

that has taken place on this subject since strong cryptographic algorithms first appeared

in the public domain, nor have the debates yet reached many definitive conclusions. Our

aim is merely to give the reader some awareness of this ongoing debate.

11.5.1 Performance of cryptographic algorithms

Figure 11.13 compares the speeds of the symmetric encryption algorithms and secure

digest functions that we have discussed in this chapter. Where available, we give two

speed measurements. In the column labelled PRB optimized we give figures based on

those published by Preneel et al. [1998]. The figures in the column labelled Crypto++

were obtained much more recently by the authors of the Crypto++ open source library

of cryptographic schemes [www.cryptopp.com]. The column headings indicate the

speed of the hardware used for these benchmarks. The PRB implementations were handoptimized assembler programs whereas the Crypto++ ones were C++ programs

generated with an optimizing compiler.

The key lengths give an indication of computational cost of a brute-force attack

on the key; the true strength of cryptographic algorithms is much more difficult to

evaluate and rests on reasoning about the success of the algorithm in obscuring the plain

text. Preneel et al. provide a useful discussion on the strength and performance of the

main symmetric algorithms.

What do these performance figures signify for real applications of cryptography,

such as their use in the TLS scheme for secure web interactions (the https protocol,

described in Section 11.6.3)? Web pages are seldom larger than 100 kilobytes, so the

contents of a page can be encrypted using any of the symmetric algorithms in a few

milliseconds, even with a processor that is quite slow by today’s standards. RSA is used

primarily for digital signatures, and that step can also be performed in a few

milliseconds. Thus the impact of algorithm performance on the perceived speed of the

https application is minimal.

Figure 11.13 Performance of symmetric encryption and secure digest algorithms

Key size/hash size

(bits)

PRB optimized

90 MHz Pentium 1

(Mbytes/s)

Crypto++

2.1 GHz Pentium 4

(Mbytes/s)

TEA 128 – 23.801

DES 56 2.113 21.340

Triple-DES 112 0.775 9.848

IDEA 128 1.219 18.963

AES 128 – 61.010

AES 192 – 53.145

AES 256 – 48.229

MD5 128 17.025 216.674

SHA-1 160 – 67.977518 CHAPTER 11 SECURITY

Asymmetric algorithms such as RSA are seldom used for data encryption, but

their performance for signing is of interest. The Crypto++ library pages indicate that

with the hardware mentioned in the last column of Figure 11.13 it takes about 4.75 ms

using RSA with a 1024-bit key to sign a secure hash (presumably using 160-bit SHA-1)

and about 0.18 ms to verify the signature.

11.5.2 Applications of cryptography and political obstacles

The algorithms described above all emerged during the 1980s and 1990s, when

computer networks were beginning to be used for commercial purposes and it was

becoming evident that their lack of security was a major problem. As we mentioned in

the introduction to this chapter, the emergence of cryptographic software was strongly

resisted by the US government. The resistance had two sources: the US National

Security Agency (NSA), which was thought to have a policy to restrict the strength of

cryptography available to other nations to a level at which the NSA could decrypt any

secret communication for military intelligence purposes; and the US Federal Bureau of

Investigation (FBI) which aimed to ensure that its agents could have privileged access

to the cryptographic keys used by all private organizations and individuals in the US for

law-enforcement purposes.

Cryptographic software was classified as a munition in the United States and was

subject to stringent export restrictions. Other countries, especially allies of the US,

applied similar (or in some cases even more stringent) restrictions. The problem was

compounded by the general ignorance among politicians and the general public as to

what cryptographic software was and its potential non-military applications. US

software companies protested that the restrictions were inhibiting the export of software

such as browsers, and the export restrictions were eventually formulated in a form that

allowed the export of code using keys of no more than 40 bits – hardly strong

cryptography!

The export restrictions may have hindered the growth of electronic commerce, but

they were not particularly effective in preventing the spread of cryptographic expertise

or in keeping cryptographic software out of the hands of users in other countries, since

many programmers inside and outside the US were eager and able to implement and

distribute cryptographic code. The current position is that software that implements

most of the major cryptographic algorithms has been available world-wide for several

years, in print [Schneier 1996] and online, in commercial and freeware versions

[www.rsasecurity.com I, cryptography.org, privacy.nb.ca, www.openssl.org].

An example is the program called PGP (Pretty Good Privacy) [Garfinkel 1994,

Zimmermann 1995], originally developed by Philip Zimmermann and carried forward

by him and others. This is part of a technical and political campaign to ensure that the

availability of cryptographic methods is not controlled by the US government. PGP has

been developed and distributed with the aim of enabling all computer users to enjoy the

level of privacy and integrity afforded by the use of public-key cryptography in their

communications. PGP generates and manages public and secret keys on behalf of a user.

It uses RSA public-key encryption for authentication and to transmit secret keys to the

intended communication partner, and it uses the IDEA or 3DES secret-key encryption

algorithms to encrypt mail messages and other documents. (At the time PGP was first

developed, use of the DES algorithm was controlled by the US government.) PGP isSECTION 11.6 CASE STUDIES: NEEDHAM–SCHROEDER, KERBEROS, TLS, 802.11 WIFI 519

widely available in both free and commercial versions. It is distributed via separate

distribution sites for North American users [www.pgp.com] and those in other parts of

the world [International PGP] to circumvent (perfectly legally) the US export

regulations.

The US government eventually recognized the futility of the NSA’s position and

the harm that it was causing to the US computer industry (which was unable to market

secure versions of web browsers, distributed operating systems and many other products

world-wide). In January 2000 the US government introduced a new policy

[www.bxa.doc.gov] intended to allow US software vendors to export software that

incorporates strong encryption. But a legal bar was retained on delivery to certain

countries and end-users, see www.rsa.com for further information. Of course, the US

does not have a monopoly on the production or the publication of cryptographic

software; open source implementations are available for all the well-known algorithms

[www.cryptopp.com]. The effect of the regulations is simply to hamper the marketing

of some US-produced commercial software products.

Other political initiatives have aimed to maintain control over the use of

cryptography by introducing legislation insisting on the inclusion of loopholes or trap

doors available only to government law-enforcement and security agencies. Such

proposals spring from the perception that secret communication channels can be very

useful to criminals of all sorts. Before the advent of digital cryptography, governments

always had the means to intercept and analyze communications between members of the

public. Strong digital cryptography radically alters that situation. But these proposals to

legislate to prevent the use of strong, uncompromized cryptography have been strongly

resisted by citizens and civil liberties bodies, who are concerned about their impact on

citizens’ privacy rights. So far, none of these legislative proposals has been adopted, but

political efforts are continuing and the eventual introduction of a legal framework for

the use of cryptography may be inevitable.

11.6 Case studies: Needham–Schroeder, Kerberos, TLS, 802.11 WiFi

The authentication protocols originally published by Needham and Schroeder [1978] are

at the heart of many security techniques. We present them in detail in Section 11.6.1.

One of the most important applications of their secret-key authentication protocol is the

Kerberos system [Neuman and Ts’o 1994], which is the subject of our second case study

(Section 11.6.2). Kerberos was designed to provide authentication between clients and

servers in networks that form a single management domain (intranets).

Our third case study (Section 11.6.3) deals with the Transport Layer Security

(TLS) protocol. This was designed specifically to meet the need for secure electronic

transactions. It is now supported by most web browsers and servers and is employed in

most of the commercial transactions that take place via the Web.

Our final case study (Section 11.6.4) illustrates the difficulty of engineering

secure systems. The IEEE 802.11 WiFi standard was published in 1999 with a security

specification included. But subsequent analysis and attacks have shown the

specification to be severely inadequate. We identify the weaknesses and relate them to

the cryptographic principles covered in this chapter.520 CHAPTER 11 SECURITY

11.6.1 The Needham–Schroeder authentication protocol

The protocols described here were developed in response to the need for a secure means

to manage keys (and passwords) in a network. At the time the work was published

[Needham and Schroeder 1978], network file services were just emerging and there was

an urgent need for better ways to manage security in local networks.

In networks that are integrated for management purposes, this need can be met by

a secure key service that issues session keys in the form of challenges (see Section

11.2.2). That is the purpose of the secret-key protocol developed by Needham and

Schroeder. In the same paper, Needham and Schroeder also set out a protocol based on

the use of public keys for authentication and key distribution that does not depend upon

the existence of secure key servers and is hence more suitable for use in networks with

many independent management domains, such as the Internet. We do not describe the

public-key version here, but the TLS protocol described in Section 11.6.3 is a variation

of it.

Needham and Schroeder proposed a solution to authentication and key

distribution based on an authentication server that supplies secret keys to clients. The

job of the authentication server is to provide a secure way for pairs of processes to obtain

shared keys. To do this, it must communicate with its clients using encrypted messages.

Needham and Schroeder with secret keys • In their model, a process acting on behalf of

a principal A that wishes to initiate secure communication with another process acting

on behalf of a principal B can obtain a key for this purpose. The protocol is described

for two arbitrary processes A and B, but in client-server systems, A is likely to be a client

initiating a sequence of requests to some server B. The key is supplied to A in two forms:

one that A can use to encrypt the messages that it sends to B and one that it can transmit

securely to B. (The latter is encrypted in a key that is known to B but not to A, so that B

can decrypt it and the key is not compromised during transmission.)

The authentication server S maintains a table containing a name and a secret key

for each principal known to the system. The secret key is used only to authenticate client

processes to the authentication server and to transmit messages securely between client

processes and the authentication server. It is never disclosed to third parties and it is

transmitted across the network at most once, when it is generated. (Ideally, a key should

always be transmitted by some other means, such as on paper or in a verbal message,

avoiding any exposure on the network.) A secret key is the equivalent of the password

used to authenticate users in centralized systems. For human principals, the name held

by the authentication service is their username and the secret key is their password. Both

are supplied by the user on request to client processes acting on the user’s behalf.

The protocol is based on the generation and transmission of tickets by the

authentication server. A ticket is an encrypted message containing a secret key for use

in communication between A and B. We tabulate the messages in the Needham and

Schroeder secret-key protocol in Figure 11.14. The authentication server is S.

NA and NB are nonces. A nonce is an integer value that is added to a message to

demonstrate its freshness. Nonces are used only once and are generated on demand. For

example, the nonces may be generated as a sequence of integer values or by reading the

clock at the sending machine.

If the protocol is successfully completed, both A and B can be sure that any

message encrypted in KAB that they receive comes from the other, and that any messageSECTION 11.6 CASE STUDIES: NEEDHAM–SCHROEDER, KERBEROS, TLS, 802.11 WIFI 521

encrypted in KAB that they send can be understood only by the other or by S (and S is

assumed to be trustworthy). This is so because the only messages that have been sent

containing KAB were encrypted in A’s secret key or B’s secret key.

There is a weakness in this protocol in that B has no reason to believe that message

3 is fresh. An intruder who manages to obtain the key KAB and make a copy of the ticket

and authenticator {KAB, A}KB (both of which might have been left in an exposed storage

location by a careless or a failed client program running under A’s authority) can use

them to initiate a subsequent exchange with B, impersonating A. For this attack to occur

an old value of KAB has to be compromised; in today’s terminology, Needham and

Schroeder did not include this possibility on their threat list, and the consensus of

opinion is that one should do so. The weakness can be remedied by adding a nonce or

timestamp to message 3, so that it becomes: {KAB, A,t}KBpub. B decrypts this message

and checks that t is recent. This is the solution adopted in Kerberos.

11.6.2 Kerberos

Kerberos was developed at MIT in the 1980s [Steiner et al. 1988] to provide a range of

authentication and security facilities for use in the campus computing network at MIT

and other intranets. It has undergone several revisions and enhancements in the light of

experience and feedback from user organizations. Kerberos version 5 [Neuman and Ts’o

1994], which we describe here, is an Internet standard (see RFC 4120 [Neuman et al.

2005]) and is used by many companies and organizations. Source code for an

implementation of Kerberos is available from MIT [web.mit.edu I]; it is included in the

OSF Distributed Computing Environment (DCE) [OSF 1997] and as the default

authentication service in Microsoft Windows [www.microsoft.com II]. An extension

Figure 11.14 The Needham–Schroeder secret-key authentication protocol

Header Message Notes

1. A → S: A, B, NA A requests S to supply a key for communication

with B.

2. S → A: {NA, B, KAB,

{KAB, A}KB}KA

S returns a message encrypted in A’s secret key,

containing a newly generated key KAB, and a

‘ticket’ encrypted in B’s secret key. The nonce NA

demonstrates that the message was sent in response

to the preceding one. A believes that S sent the

message because only S knows A’s secret key.

3. A → B: {KAB, A}KB A sends the ticket to B.

4. B → A: {NB}KAB B decrypts the ticket and uses the new key, KAB, to

encrypt another nonce, NB.

5. A → B: {NB – 1}KAB A demonstrates to B that it was the sender of the

previous message by returning an agreed

transformation of NB.522 CHAPTER 11 SECURITY

was included to incorporate the use of public-key certificates for the initial

authentication of principals (Step A in Figure 11.15) [Neuman et al. 1999].

Figure 11.15 shows the process architecture. Kerberos deals with three kinds of

security object:

Ticket: A token issued to a client by the Kerberos ticket-granting service for

presentation to a particular server, verifying that the sender has recently been

authenticated by Kerberos. Tickets include an expiry time and a newly generated

session key for use by the client and the server.

Authenticator: A token constructed by a client and sent to a server to prove the

identity of the user and the currency of any communication with a server. An

authenticator can be used only once. It contains the client’s name and a timestamp

and is encrypted in the appropriate session key.

Session key: A secret key randomly generated by Kerberos and issued to a client for

use when communicating with a particular server. Encryption is not mandatory for

all communication with servers; the session key is used for encrypting

communication with those servers that demand it and for encrypting all

authenticators (see above).

Client processes must possess a ticket and a session key for each server that they use. It

would be impractical to supply a new ticket and key for each client-server interaction,

so most tickets are granted to clients with a lifetime of several hours so that they can be

used for interaction with a particular server until they expire.

A Kerberos server is known as a Key Distribution Centre (KDC). Each KDC

offers an Authentication Service (AS) and a Ticket-Granting Service (TGS). On login,

Figure 11.15 System architecture of Kerberos

Client Server

DoOperation

•••

Authentication

database

Login

session setup

Ticketgranting

service T

Kerberos Key Distribution Centre

Server

session setup

Authentication

service A

1. Request for

TGS ticket

2. TGS

ticket

3. Request for

server ticket

4. Server ticket

5. Service

request

Request encrypted with session key

Reply encrypted with session key

Service

function

Step B

Step A

Step C

C SSECTION 11.6 CASE STUDIES: NEEDHAM–SCHROEDER, KERBEROS, TLS, 802.11 WIFI 523

users are authenticated by the AS, using a network-secure variation of the password

method, and the client process acting on behalf of the user is supplied with a ticketgranting ticket and a session key for communicating with the TGS. Subsequently, the

original client process and its descendants can use the ticket-granting ticket to obtain

tickets and session keys for specific services from the TGS.

The Needham and Schroeder [1978] protocol is followed quite closely in

Kerberos, with time values (integers representing a date and time) used as nonces. This

serves two purposes:

• to guard against replay of old messages intercepted in the network or the reuse of

old tickets found lying in the memory of machines from which the authorized user

has logged out (nonces were used to achieve this purpose in Needham and

Schroeder);

• to apply a lifetime to tickets, enabling the system to revoke users’ rights when, for

example, they cease to be authorized users of the system.

Below we describe the Kerberos protocols in detail, using the notation defined at the

bottom of the page. First, we describe the protocol by which the client obtains a ticket

and a session key for access to the TGS.

A Kerberos ticket has a fixed period of validity starting at time t1 and ending at

time t2. A ticket for a client C to access a server S takes the form:

, which we denote as

The client’s name is included in the ticket to avoid possible use by impostors, as we shall

see later. The step and message numbers in Figure 11.15 correspond to those in tabulated

description A. Note that message 1 is not encrypted and does not include C’s password.

It contains a nonce that is used to check the validity of the reply.

A. Obtain Kerberos session key and TGS ticket, once per login session

Header Message Notes

1. C → A:

Request for

TGS ticket

C, T, n Client C requests the Kerberos

authentication server A to supply a

ticket for communication with the

ticket-granting service T.

2. A → C:

TGS session

key and

ticket

{KCT, n}KC, {ticket(C,T)}KT A returns a message containing a

ticket encrypted in its secret key

and a session key for C to use with

T. The inclusion of the nonce n

encrypted in KC shows that the

message comes from the recipient

of message 1, who must know KC.

Notation:

A Name of Kerberos authentication service.

T Name of Kerberos ticket-granting service.

C Name of client.

n A nonce.

t A timestamp.

t1 Starting time for validity of ticket.

t2 Ending time for validity of ticket.

C S t

{ } , , , , 1 t2 KCS K

S

{ } ticket C S ( ) , K

S

containing

C, T, t1, t2, KCT524 CHAPTER 11 SECURITY

Message 2 is sometimes called a ‘challenge’ because it presents the requester with

information that is only useful if it knows C’s secret key, KC. An impostor who attempts

to impersonate C by sending message 1 can get no further, since they cannot decrypt

message 2. For principals that are users, KC is a scrambled version of the user’s

password. The client process will prompt the user to type their password and will

attempt to decrypt message 2 with it. If the user gives the right password, the client

process obtains the session key KCT and a valid ticket for the ticket-granting service; if

not, it obtains gibberish. Servers have secret keys of their own, known only to the

relevant server process and to the authentication server.

When a valid ticket has been obtained from the authentication service, the client

C can use it to communicate with the ticket-granting service to obtain tickets for other

servers any number of times until the ticket expires. Thus to obtain a ticket for any server

S, C constructs an authenticator encrypted in KCT of the form:

{C, t}KCT, which we denote as {auth(C)}KCT, and sends a request to T:

C is then ready to issue request messages to the server, S:

For the client to be sure of the server’s authenticity, S should return the nonce n to C (to

reduce the number of messages required, this could be included in the messages that

contain the server’s reply to the request):

B. Obtain ticket for a server S, once per client-server session

3. C → T:

Request

ticket for

service S

{auth(C)}KCT ,

{ticket(C,T)}KT , S, n

C requests the ticket-granting

server T to supply a ticket for

communication with another

server, S.

4. T → C:

Service

ticket

{KCS, n}KCT , {ticket(C,S)}KS T checks the ticket. If it is valid T

generates a new random session

key, KCS, and returns it with a

ticket for S (encrypted in the

server’s secret key, KS).

C. Issue a server request with a ticket

5. C → S:

Service

request

{auth(C)}KCS , {ticket(C,S)}KS ,

request, n

C sends the ticket to S with a

newly generated authenticator for

C and a request. The request

would be encrypted in KCS if

secrecy of the data is required.

D. Authenticate server (optional)

6. S → C:

Server authentication

{n}KCS (Optional): S sends the nonce to C,

encrypted in KCS.SECTION 11.6 CASE STUDIES: NEEDHAM–SCHROEDER, KERBEROS, TLS, 802.11 WIFI 525

Application of Kerberos • Kerberos was developed for use in Project Athena at MIT – a

campus-wide networked computing facility for undergraduate education with many

workstations and servers providing a service to more than 5000 users. The environment

is such that neither the trustworthiness of clients nor the security of the network and the

machines that offer network services can be assumed – for example, workstations are

not protected against the installation of user-developed system software, and server

machines (other than the Kerberos server) are not necessarily secured against physical

interference with their software configuration.

Kerberos provides virtually all of the security in the Athena system. It is used to

authenticate users and other principals. Most of the servers running on the network have

been extended to require a ticket from each client at the start of every client-server

interaction. These include file storage (NFS and Andrew File System), electronic mail,

remote login and printing. Users’ passwords are known only to the user and to the

Kerberos authentication service. Services have secret keys that are known only to

Kerberos and the servers that provide the service.

We describe here the way in which Kerberos is applied to the authentication of

users on login. Its use to secure the NFS file service is described in Chapter 12.

Login with Kerberos • When a user logs into a workstation, the login program sends the

user’s name to the Kerberos authentication service. If the user is known to the

authentication service, it replies with a session key, a nonce encrypted in the user’s

password and a ticket for the TGS. The login program then attempts to decrypt the

session key and the nonce using the password that the user typed in response to the

password prompt. If the password is correct, the login program obtains the session key

and the nonce. It checks the nonce and stores the session key with the ticket for

subsequent use when communicating with the TGS. At this point, the login program can

erase the user’s password from its memory, since the ticket now serves to authenticate

the user. A login session is then started for the user on the workstation. Note that the

user’s password is never exposed to eavesdropping on the network – it is retained in the

workstation and is erased from memory soon after it is entered.

Accessing servers with Kerberos • Whenever a program running on a workstation needs

to access a new service, it requests a ticket for the service from the ticket-granting service. For example, when a UNIX user wishes to log into a remote computer, the rlogin

command program on the user’s workstation obtains a ticket from the Kerberos ticketgranting service for access to the rlogind network service. The rlogin command program

sends the ticket, together with a new authenticator, in a request to the rlogind process on

the computer where the user wishes to log in. The rlogind program decrypts the ticket

with the rlogin service’s secret key and checks the validity of the ticket (that is, that the

ticket’s lifetime has not expired). Server machines must take care to store their secret

keys in storage that is inaccessible to intruders.

The rlogind program then uses the session key included in the ticket to decrypt the

authenticator and checks that the authenticator is fresh (authenticators can be used only

once). Once the rlogind program is satisfied that the ticket and authenticator are valid,

there is no need for it to check the user’s name and password, because the user’s identity

is known to the rlogind program and a login session is established for that user on the

remote machine.526 CHAPTER 11 SECURITY

Implementation of Kerberos • Kerberos is implemented as a server that runs on a secure

machine. A set of libraries is provided for use by client applications and services. The

DES encryption algorithm is used, but this is implemented as a separate module that can

be easily replaced.

The Kerberos service is scalable – the world is divided into separate domains of

authentication authority, called realms, each with its own Kerberos server. Most

principals are registered in just one realm, but the Kerberos ticket-granting servers are

registered in all of the realms. Principals can authenticate themselves to servers in other

realms through their local ticket-granting server.

Within a single realm, there can be several authentication servers, all of which

have copies of the same authentication database. The authentication database is

replicated by a simple master–slave technique. Updates are applied to the master copy

by a single Kerberos Database Management service (KDBM) that runs only on the

master machine. The KDBM handles requests from users to change their passwords and

requests from system administrators to add or delete principals and to change their

passwords.

To make this scheme transparent to users, the lifetime of TGS tickets ought to be

as long as the longest possible login session, since the use of an expired ticket will result

in the rejection of service requests; the only remedy is for the user to reauthenticate the

login session and then request new server tickets for all of the services in use. In practice,

ticket lifetimes in the region of 12 hours are used.

Critiques of Kerberos • The protocol for Kerberos version 5 described above contains

several improvements designed to deal with criticisms of earlier versions [Bellovin and

Merritt 1990, Burrows et al. 1990]. The most important criticism of version 4 was that

the nonces used in authenticators were implemented as timestamps, and protection

against the replay of authenticators depended upon at least loose synchronization of

clients’ and servers’ clocks. Furthermore, if a synchronization protocol is used to bring

client and server clocks into loose synchrony, the synchronization protocol must itself

be secure against security attacks. See Chapter 14 for information on clock

synchronization protocols.

The protocol definition for version 5 allows the nonces in authenticators to be

implemented as timestamps or as sequence numbers. In both cases, it requires that they

be unique and that servers hold a list of recently received nonces from each client to

check that they are not replayed. This is an inconvenient implementation requirement

and is difficult for servers to guarantee in case of failures. Kehne et al. [1992] have

published a proposed improvement to the Kerberos protocol that does not rely on

synchronized clocks.

The security of Kerberos depends on limited session lifetimes. The period of

validity of TGS tickets is generally limited to a few hours; the period must be chosen to

be long enough to avoid inconvenient interruptions of service but short enough to ensure

that users who have been deregistered or downgraded do not continue to use the

resources for more than a short period. This might cause difficulties in some commercial

environments, because the consequent requirement for the user to supply a new set of

authentication details at an arbitrary point in the interaction might intrude on the

application.SECTION 11.6 CASE STUDIES: NEEDHAM–SCHROEDER, KERBEROS, TLS, 802.11 WIFI 527

11.6.3 Securing electronic transactions with secure sockets

The Secure Sockets Layer (SSL) protocol was originally developed by the Netscape

Corporation [www.mozilla.org] and proposed as a standard specifically to meet the

needs described below. An extended version of SSL has been adopted as an Internet

standard under the name Transport Layer Security (TLS), described in RFC 2246

[Dierks and Allen 1999]. TLS is supported by most browsers and is widely used in

Internet commerce. We explore its main features below:

Negotiable encryption and authentication algorithms • In an open network we should

not assume that all parties use the same client software or that all client and server

software includes a particular encryption algorithm. In fact, the laws of some countries

attempt to restrict the use of certain encryption algorithms to those countries alone. TLS

has been designed so that the algorithms used for encryption and authentication are

negotiated between the processes at the two ends of the connection during the initial

handshake. It may turn out that they do not have sufficient algorithms in common, and

in that case the connection attempt will fail.

Bootstrapped secure communication • To meet the need for secure communication

without previous negotiation or help from third parties, the secure channel is established

using a protocol similar to the hybrid scheme mentioned in Section 11.3.3. Unencrypted

communication is used for the initial exchanges, then public-key cryptography and

finally secret-key cryptography once a shared secret key has been established. Each

switch is optional and preceded by a negotiation.

Thus the secure channel is fully configurable, allowing communication in each

direction to be encrypted and authenticated but not requiring it, so that computing

resources need not be consumed in performing unnecessary encryption.

The details of the TLS protocol are published and standardized, and several

software libraries and toolkits are available to support it [Hirsch 1997,

www.openssl.org], some of them in the public domain. It has been incorporated in a

wide range of application software, and its security has been verified by independent

review.

Figure 11.16 TLS protocol stack

(Figures 11.16 to 11.19 are based on diagrams in Hirsch [1997] and are used with Frederick Hirsch’s permission)

TLS

Handshake

Protocol

TLS Change

Cipher Spec

TLS Alert

Protocol

Transport layer (usually TCP)

Network layer (usually IP)

TLS Record Protocol

HTTP Telnet • • • • •

TLS protocols: Other protocols:528 CHAPTER 11 SECURITY

TLS consists of two layers (Figure 11.16): TLS Record Protocol layer, which

implements a secure channel, encrypting and authenticating messages transmitted

through any connection-oriented protocol; and a handshake layer, containing the TLS

handshake protocol and two other related protocols that establish and maintain a TLS

session (that is, a secure channel) between a client and a server. Both are usually

implemented by software libraries at the application level in the client and the server.

The TLS record protocol is a session-level layer; it can be used to transport applicationlevel data transparently between a pair of processes while guaranteeing its secrecy,

integrity and authenticity. These are exactly the properties we specified for secure

channels in our security model (Section 2.4.3), but in TLS there are options for the

communicating partners to choose whether or not to deploy decryption and

authentication of messages in each direction. Each secure session is given an identifier,

and each partner can store session identifiers in a cache for subsequent reuse, avoiding

the overhead of establishing a new session when another secure session with the same

partner is required.

TLS is widely used to add a secure communication layer below existing

application-level protocols. It is probably most widely used to secure HTTP interactions

for use in Internet commerce and other security-sensitive applications. It is implemented

by virtually all web browsers and web servers: the use of the protocol prefix https: in

URLs initiates the establishment of a TLS secure channel between a browser and a web

server. It has also been widely deployed to provide secure implementations of Telnet,

FTP and many other application protocols. TLS is the de facto standard for use in

applications requiring secure channels; there is a wide choice of available

implementations, both commercial and public-domain, with APIs for CORBA and Java.

The TLS handshake protocol is illustrated in Figure 11.17. The handshake is

performed over an existing connection. It begins in the clear and it establishes a TLS

session by exchanging the agreed options and parameters needed to perform encryption

and authentication. The handshake sequence varies depending on whether client and

server authentication are required. The handshake protocol may also be invoked at a

later time to change the specification of a secure channel – for example, communication

may begin with message authentication using message authentication codes only, and at

a later point, encryption may be added. This is achieved by performing the handshake

protocol again to negotiate a new cipher specification using the existing channel.

The TLS initial handshake is potentially vulnerable to man-in-the-middle attacks,

as described in Section 11.2.2, Scenario 3. To protect against them, the public key used

to verify the first certificate received may be delivered by a separate channel – for

example, browsers and other Internet software delivered on a CD-ROM may include a

set of public keys for some well-known certificate authorities. Another defence for the

clients of well-known services is based on the inclusion of the service’s domain name in

its public-key certificates – clients should only deal with the service at the IP address

corresponding to that domain name.

TLS supports a variety of options for the cryptographic functions to be used.

These are collectively known as a cipher suite. A cipher suite includes a single choice

for each of the features shown in Figure 11.18.

A variety of popular cipher suites are preloaded, with standard identifiers in the

client and the server. During the handshake, the server offers the client a list of the cipher

suite identifiers that it has available, and the client responds by selecting one of them (orSECTION 11.6 CASE STUDIES: NEEDHAM–SCHROEDER, KERBEROS, TLS, 802.11 WIFI 529

giving an error indication if it has none that match). At this stage they also agree on a

(optional) compression method and a random start value for CBC block encryption

functions (see Section 11.3).

Next, the partners optionally authenticate each other by exchanging signed publickey certificates in X.509 format. These certificates may be obtained from a public-key

authority or they may simply be generated temporarily for the purpose. In any case, at

least one public key must be available for use in the next stage of the handshake.

One partner then generates a pre-master secret and sends it to the other partner

encrypted with the public key. A pre-master secret is a large random value that is used

by both partners to generate the two session keys (called write keys) for encrypting data

Figure 11.17 TLS handshake protocol

Client Server

ClientHello

ServerHello

Certificate

Certificate Request

ServerHelloDone

Certificate

Certificate Verify

Change Cipher Spec

Finished

Change Cipher Spec

Finished

Establish protocol version, session

ID, cipher suite, compression

method, exchange random values

Optionally send server certificate

and request client certificate

Send client certificate response if

requested

Change cipher suite and finish

handshake

Figure 11.18 TLS handshake configuration options

Component Description Example

Key exchange

method

The method to be used for

exchange of a session key

RSA with public-key

certificates

Cipher for data

transfer

The block or stream cipher to be

used for data

IDEA

Message digest

function

For creating message

authentication codes (MACs)

SHA-1530 CHAPTER 11 SECURITY

in each direction and the message authentication secrets to be used for message

authentication. When all this has been done, a secure session begins. This is triggered

by the ChangeCipherSpec messages exchanged between the partners. These are

followed by Finished messages. Once the Finished messages have been exchanged, all

further communication is encrypted and signed according to the chosen cipher suite with

the agreed keys.

Figure 11.19 shows the operation of the record protocol. A message for

transmission is first fragmented into blocks of a manageable size, then the blocks are

optionally compressed. Compression is not strictly a feature of secure communication,

but it is provided here because a compression algorithm can usefully share some of the

work of processing the bulk data with the encryption and digital signature algorithms.

In other words, a pipeline of data transformations can be set up within the TLS record

layer that will perform all of the transformations required more efficiently than could be

done independently.

The encryption and message authentication (MAC) transformations deploy the

algorithms specified in the agreed cipher suite exactly as described in Sections 11.3.1

and 11.4.2. Finally, the signed and encrypted block is transmitted to the partner through

the associated TCP connection, where the transformations are reversed to produce the

original data block.

Summary • TLS provides a practical implementation of a hybrid encryption scheme

with authentication and key exchange based on public keys. Because the ciphers are

negotiated in the handshake, it does not depend upon the availability of any particular

algorithms. Nor does it depend upon any secure services at the time of session

abcdefghi

abc def ghi

Figure 11.19 TLS record protocol

Application data

Record protocol units

Compressed units

MAC

Encrypted

TCP packet

Fragment/combine

Compress

Hash

Encrypt

TransmitSECTION 11.6 CASE STUDIES: NEEDHAM–SCHROEDER, KERBEROS, TLS, 802.11 WIFI 531

establishment. The only requirement is that the public-key certificates are issued by an

authority recognized by both parties.

Because the SSL basis for TLS and its reference implementation were published

[www.mozilla.org], it was the subject of review and debate. Some amendments were

made to the early designs, and it was widely endorsed as a valuable standard. TLS is now

integrated in most web browsers and web servers and is used in other applications such

as secure Telnet and FTP. Commercial and public-domain [www.rsasecurity.com I,

Hirsch 1997, www.openssl.org] implementations are widely available in the form of

libraries and browser plug-ins.

11.6.4 Weaknesses in the original IEEE 802.11 WiFi security design

The IEEE 802.11 standard for wireless LANs described in Section 3.5.2 was first

released in 1999 [IEEE 1999]. It was implemented in base stations, laptops and portable

devices from a similar date and widely used for mobile communication. Unfortunately,

the security design in the standard was subsequently found to be severely inadequate in

several respects. We outline that initial design and its weaknesses as a case study in the

difficulties of security design already referred to in Section 11.1.3.

It was recognized that wireless networks are by their nature more exposed to

attack than wired networks because the network and the transmitted data are available

for eavesdropping and masquerading by any device equipped with a transmitter/receiver

within range. The initial 802.11 design therefore aimed to provide access control for

WiFi networks and privacy and integrity for the data transmitted on them through a

security specification entitled Wired Equivalent Privacy (WEP), which embodies the

following measures, that can optionally be activated by a network administrator:

Access control by a challenge-response protocol (cf. Kerberos, Section 11.6.2), in

which a joining node is challenged by the base station to demonstrate that it has the

correct shared key. A single key, K, is assigned by a network administrator and

shared between the base station and all authorized devices.

Privacy and integrity using an optional encryption mechanism based on the RC4

stream cipher. The same key, K, used for access control is also used in encryption.

There are key length options of 40, 64 or 128 bits. An encrypted checksum is

included in each packet to protect its integrity.

The following deficiencies and design weaknesses were discovered soon after the

standard was deployed:

The sharing of a single key by all users of a network renders the design weak in

practice, since:

– The key is liable to be transmitted to new users on unprotected channels.

– A single careless or malicious user (such as a disgruntled former employee)

who has gained access to the key can compromise the security of the entire

network, and this can go undiscovered.

Solution: Use a public-key-based protocol for negotiating individual keys, as is

done in TLS/SSL (see Section 11.6.3).532 CHAPTER 11 SECURITY

Base stations are never authenticated, so an attacker who knows the current shared

key could introduce a spoof base station and eavesdrop on, insert or tamper with any

traffic.

Solution: Base stations should supply a certificate that can be authenticated by

the use of a public key obtained from a third party.

Inappropriate use of a stream cipher rather than a block cipher (see descriptions of

block and stream ciphers in Section 11.3). Figure 11.20 shows the process of

encryption and decyption in 802.11 WEP security. Each packet is encrypted by

XOR-ing its content with a keystream produced by the RC4 algorithm. The receiving

station uses RC4 to generate the same keystream and decrypts the packet with

another XOR. To avoid keystream synchronization errors when packets are lost or

corrupted, RC4 is restarted with a start value consisting of a 24-bit initial value

concatenated with the globally shared key. The initial value is updated and included

(in the clear) in each packet transmitted. The shared key cannot easily be changed in

normal use, so the starting value has only s = 224 (or about 107) different states,

resulting in the repetition of the start value and hence the keystream, after 107 packets

are sent. In practice this can occur within a few hours, and even shorter repetition

cycles can arise if packets are lost. An attacker receiving the encrypted packets can

always detect repetitions since the initial value is sent in the clear.

The RC4 specification explicitly warns against keystream repetition. This is

because an attacker who receives an encrypted packet Ci and knows the plain text Pi

(for example, by guessing that it is a standard enquiry to a server) can calculate the

keystream Ki used to encrypt the packet. The same value of Ki will recur after s

packets are transmitted, and the attacker can use it to decrypt the newly transmitted

Figure 11.20 Use of RC4 stream cipher in IEEE 802.11 WEP

K

Encryption

IV: initial value

K: shared key

IV

plaintext

RC4

Decryption

Increment

XOR

keystream

cipher text IV

K

IV

RC4

cipher text IV XOR plaintextSECTION 11.6 CASE STUDIES: NEEDHAM–SCHROEDER, KERBEROS, TLS, 802.11 WIFI 533

packet. The attacker may eventually succeed in decrypting a high proportion of

packets in this manner by guessing plaintext packets correctly. This weakness was

first pointed out by Borisov et al. [2001] and led to a major reappraisal of WEP

security and its replacement in later versions of 802.11.

Solution: Negotiate a new key after a time less than the worst case for repetition.

An explicit termination code would be needed, as is the case in TLS.

Key lengths of 40 bits and 64 bits were included in the standard to enable products to

be shipped abroad by US suppliers at a time when the US government regulations

referred to in Section 11.5.2 restricted key lengths for exported devices to 40 bits (and

subsequently 64 bits). But 40-bit keys are so easily cracked by brute force that they

offer very little security, and even 64-bit keys are potentially crackable with a

determined attack.

Solution: Use 128-bit keys only. This has been adopted in many recent WiFi

products.

The RC4 stream cipher was shown, after publication of the 802.11 standard, to have

weaknesses that enabled the key to be discovered after observation of a substantial

quantity of traffic even without repetition of the keystream [Fluhrer et al. 2001]. This

weakness was demonstrated in practice. It rendered the WEP scheme insecure even

with 128-bit keys and led some companies to ban the use of WiFi networks by their

employees.

Solution: Provide for the negotiation of cipher specifications as is done in TLS,

giving a choice of encryption algorithms. RC4 is hard-wired into the WEP

standard, with no provision for the negotiation of encryption algorithms.

Users often didn’t deploy the protection offered by the WEP scheme, probably

because they didn’t realize just how exposed their data was. This was not a weakness

in the design of the standard but in the marketing of products based on it. Most were

designed to start up with security disabled and their documentation of the security

risks was often weak.

Solution: Better default settings and documentation can help. Users want to

obtain optimum performance, though, and communication was perceptibly slower

with encryption enabled using the hardware available at the time. Attempts to

avoid the use of WEP encryption led to the addition to base stations of features for

the suppression of the identifying packets normally broadcast by base stations and

the rejection of packets not sent from an authorized MAC address (see Section

3.5.1). Neither of these offered much security, since a network can be discovered

by intercepting (‘sniffing’) packets in transmission and MAC addresses can be

spoofed by operating system modifications.

IEEE set up a dedicated task group to create a replacement security solution and their

work led to a completely new security protocol known as Wi-Fi Protected Access

(WPA). This was specified in the IEEE 802.11i draft [IEEE 2004b, Edney and Arbaugh

2003], and started to appear in products in mid-2003. IEEE 802.11i (also known as

WPA2) itself was ratified in June 2004, and uses AES encryption, instead of RC4, which

was used in WEP. Subsequent developments of IEEE 802.11 also incorporate WPA2.534 CHAPTER 11 SECURITY

11.7 Summary

Threats to the security of distributed systems are pervasive. It is essential to protect the

communication channels and the interfaces of any system that handles information that

could be the subject of attacks. Personal mail, electronic commerce and other financial

transactions are all examples of such information. Security protocols are carefully

designed to guard against loopholes. The design of secure systems starts from a list of

threats and a set of ‘worst case’ assumptions.

Security mechanisms are based on public-key and secret-key cryptography.

Cryptographic algorithms scramble messages in a manner that cannot be reversed

without knowledge of the decryption key. Secret-key cryptography is symmetric – the

same key serves for both encryption and decryption. If two parties share a secret key,

they can exchange encrypted information without risk of eavesdropping or tampering

and with guarantees of authenticity.

Public-key cryptography is asymmetric – separate keys are used for encryption

and decryption, and knowledge of one does not reveal the other. One key is made public,

enabling anyone to send secure messages to the holder of the corresponding private key

and allowing the holder of the private key to sign messages and certificates. Certificates

can act as credentials for the use of protected resources.

Resources are protected by access-control mechanisms. Access-control schemes

assign rights to principals (that is, the holders of credentials) to perform operations on

distributed objects and collections of objects. Rights may be held in access control lists

(ACLs) associated with collections of objects or they may be held by principals in the

form of capabilities – unforgeable keys for access to collections of resources.

Capabilities are convenient for the delegation of access rights but are hard to revoke.

Changes to ACLs take effect immediately, revoking the previous access rights, but they

are more complex and costly to manage than capabilities.

Until recently, the DES encryption algorithm was the most widely used symmetric

encryption scheme, but its 56-bit keys are no longer safe against brute-force attacks. The

triple version of DES gives 112-bit key strength, which is safe, but other modern

algorithms (such as IDEA and AES) are much faster and provide greater strength.

RSA is the most widely used asymmetric encryption scheme. For safety against

factoring attacks, it should be used with 768-bit keys or greater. Public-key

(asymmetric) algorithms are outperformed by secret-key (symmetric) algorithms by

several orders of magnitude, so they are generally used only in hybrid protocols such as

TLS, for the establishment of secure channels that use shared keys for subsequent

exchanges.

The Needham–Schroeder authentication protocol was the first general-purpose,

practical security protocol, and it still provides the basis for many practical systems.

Kerberos is a well-designed scheme for the authentication of users and the protection of

services within a single organization. Kerberos is based on Needham–Schroeder and

symmetric cryptography. TLS is the security protocol designed for and used widely in

electronic commerce. It is a flexible protocol for the establishment and use of secure

channels based on both symmetric and asymmetric cryptography. The weaknesses of

IEEE 802.11 WiFi security provide an object lesson in the difficulties of security design.EXERCISES 535

EXERCISES

11.1 Describe some of the physical security policies in your organization. Express them in

terms that could be implemented in a computerized door locking system. page 480

11.2 Describe some of the ways in which conventional email is vulnerable to eavesdropping,

masquerading, tampering, replay and denial of service attacks. Suggest methods by

which email could be protected against each of these forms of attack. page 482

11.3 Initial exchanges of public keys are vulnerable to man-in-the-middle attacks. Describe

as many defences against it as you can. pages 489, 527

11.4 PGP is often used to secure email communication. Describe the steps that a pair of users

using PGP must take before they can exchange email messages with privacy and

authenticity guarantees. What scope is there to make the preliminary key negotiation

invisible to users? (The PGP negotiation is an instance of the hybrid scheme.)

pages 509, 518

11.5 How would email be sent to a large list of recipients using PGP or a similar scheme?

Suggest a scheme that is simpler and faster when the list is used frequently.

page 518, Section 4.4

11.6 The implementation of the TEA symmetric encryption algorithm given in Figures 11.7–

11.9 is not portable between all machine architectures. Explain why. How could a

message encrypted using the TEA implementation be transmitted to decrypt it correctly

on all other architectures? page 504

11.7 Modify the TEA application program in Figure 11.9 to use cipher block chaining (CBC).

pages 501, 504

11.8 Construct a stream cipher application based on the program in Figure 11.9.

pages 502, 504

11.9 Estimate the time required to crack a 56-bit DES key by a brute-force attack using a

2000 MIPS (million instruction per second) computer, assuming that the inner loop for

a brute-force attack program involves around 10 instructions per key value, plus the time

to encrypt an 8-byte plaintext (see Figure 11.13). Perform the same calculation for a

128-bit IDEA key. Extrapolate your calculations to obtain the cracking time for a

200,000 MIPS parallel processor (or an Internet consortium with similar processing

power).

page 505

11.10 In the Needham and Shroeder authentication protocol with secret keys, explain why the

following version of message 5 is not secure:

A → B: {NB}KAB

page 520

11.11 Review the solutions proposed in the discussion of the 802.11 Wireless Equivalent

Privacy protocol design, outlining ways in which each solution could be implemented

and mentioning any drawbacks or inconveniences. (5 answers) page 531This page intentionally left blank537

12

DISTRIBUTED FILE SYSTEMS

12.1 Introduction

12.2 File service architecture

12.3 Case study: Sun Network File System

12.4 Case study: The Andrew File System

12.5 Enhancements and further developments

12.6 Summary

A distributed file system enables programs to store and access remote files exactly as they

do local ones, allowing users to access files from any computer on a network. The

performance and reliability experienced for access to files stored at a server should be

comparable to that for files stored on local disks.

In this chapter we define a simple architecture for file systems and describe two

basic distributed file service implementations with contrasting designs that have been in

widespread use for over two decades:

• the Sun Network File System, NFS;

• the Andrew File System, AFS.

Each emulates the UNIX file system interface, with differing degrees of scalability, fault

tolerance and deviation from the strict UNIX one-copy file update semantics.

Several related file systems that exploit new modes of data organization on disk or

across multiple servers to achieve high-performance, fault-tolerant and scalable file

systems are also reviewed. Other types of distributed storage system are described

elsewhere in the book. These include peer-to-peer storage systems (Chapter 10),

replicated file systems (Chapter 18), multimedia data servers (Chapter 20) and the

particular style of storage service required to support Internet search and other largescale, data-intensive applications (Chapter 21).538 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

12.1 Introduction

In Chapters 1 and 2, we identified the sharing of resources as a key goal for distributed

systems. The sharing of stored information is perhaps the most important aspect of

distributed resource sharing. Mechanisms for data sharing take many forms and are

described in several parts of this book. Web servers provide a restricted form of data

sharing in which files stored locally, in file systems at the server or in servers on a local

network, are made available to clients throughout the Internet. The design of large-scale

wide area read-write file storage systems poses problems of load balancing, reliability,

availability and security, whose resolution is the goal of the peer-to-peer file storage

systems described in Chapter 10. Chapter 18 focuses on replicated storage systems that

are suitable for applications requiring reliable access to data stored on systems where the

availability of individual hosts cannot be guaranteed. In Chapter 20 we describe a media

server that is designed to serve streams of video data to large numbers of users in real

time. Chapter 21 describes a file system designed to support large-scale, data-intensive

applications such as Internet search.

The requirements for sharing within local networks and intranets lead to a need

for a different type of service – one that supports the persistent storage of data and

programs of all types on behalf of clients and the consistent distribution of up-to-date

data. The purpose of this chapter is to describe the architecture and implementation of

these basic distributed file systems. We use the word ‘basic’ here to denote distributed

file systems whose primary purpose is to emulate the functionality of a non-distributed

file system for client programs running on multiple remote computers. They do not

maintain multiple persistent replicas of files, nor do they support the bandwidth and

timing guarantees required for multimedia data streaming – those requirements are

addressed in later chapters. Basic distributed file systems provide an essential

underpinning for organizational computing based on intranets.

File systems were originally developed for centralized computer systems and

desktop computers as an operating system facility providing a convenient programming

interface to disk storage. They subsequently acquired features such as access-control

and file-locking mechanisms that made them useful for the sharing of data and

programs. Distributed file systems support the sharing of information in the form of files

and hardware resources in the form of persistent storage throughout an intranet. A welldesigned file service provides access to files stored at a server with performance and

reliability similar to, and in some cases better than, files stored on local disks. Their

design is adapted to the performance and reliability characteristics of local networks,

and hence they are most effective in providing shared persistent storage for use in

intranets. The first file servers were developed by researchers in the 1970s [Birrell and

Needham 1980, Mitchell and Dion 1982, Leach et al. 1983], and Sun’s Network File

System became available in the early 1980s [Sandberg et al. 1985, Callaghan 1999].

A file service enables programs to store and access remote files exactly as they do

local ones, allowing users to access their files from any computer in an intranet. The

concentration of persistent storage at a few servers reduces the need for local disk

storage and (more importantly) enables economies to be made in the management and

archiving of the persistent data owned by an organization. Other services, such as the

name service, the user authentication service and the print service, can be more easilySECTION 12.1 INTRODUCTION 539

implemented when they can call upon the file service to meet their needs for persistent

storage. Web servers are reliant on filing systems for the storage of the web pages that

they serve. In organizations that operate web servers for external and internal access via

an intranet, the web servers often store and access the material from a local distributed

file system.?

With the advent of distributed object-oriented programming, a need arose for the

persistent storage and distribution of shared objects. One way to achieve this is to

serialize objects (in the manner described in Section 4.3.2) and to store and retrieve the

serialized objects using files. But this method for achieving persistence and distribution

is impractical for rapidly changing objects, so several more direct approaches have been

developed. Java remote object invocation and CORBA ORBs provide access to remote,

shared objects, but neither of these ensures the persistence of the objects, nor are the

distributed objects replicated.

Figure 12.1 provides an overview of types of storage system. In addition to those

already mentioned, the table includes distributed shared memory (DSM) systems and

persistent object stores. DSM was described in Chapter 6. It provides an emulation of a

shared memory by the replication of memory pages or segments at each host, but it does

not necessarily provide automatic persistence. Persistent object stores were introduced

in Chapter 5. They aim to provide persistence for distributed shared objects. Examples

include the CORBA Persistent State Service (see Chapter 8) and persistent extensions

to Java [Jordan 1996, java.sun.com VIII]. Some research projects have developed in

platforms that support the automatic replication and persistent storage of objects (for

example, PerDiS [Ferreira et al. 2000] and Khazana [Carter et al. 1998]). Peer-to-peer

storage systems offer scalability to support client loads much larger than the systems

described in this chaper, but they incur high performance costs in providing secure

access control and consistency between updatable replicas.

Figure 12.1 Storage systems and their properties

Types of consistency:

1: strict one-copy : slightly weaker guarantees 2: considerably weaker guarantees

Sharing Persistence Distributed

cache/replicas

Consistency

maintenance

Example

Main memory 1 RAM

File system 1 UNIX file system

Distributed file system Sun NFS

Web Web server

Distributed shared memory Ivy (DSM, Ch. 6)

Remote objects (RMI/ORB) 1 CORBA

Persistent object store 1 CORBA Persistent

State Service

Peer-to-peer storage system 2 OceanStore (Ch. 10)540 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

The consistency column indicates whether mechanisms exist for the maintenance

of consistency between multiple copies of data when updates occur. Virtually all storage

systems rely on the use of caching to optimize the performance of programs. Caching

was first applied to main memory and non-distributed file systems, and for those the

consistency is strict (denoted by a ‘1’, for one-copy consistency in Figure 12.1) –

programs cannot observe any discrepancies between cached copies and stored data after

an update. When distributed replicas are used, strict consistency is more difficult to

achieve. Distributed file systems such as Sun NFS and the Andrew File System cache

copies of portions of files at client computers, and they adopt specific consistency

mechanisms to maintain an approximation to strict consistency – this is indicated by a

tick ( ) in the consistency column of Figure 12.1. We discuss these mechanisms and

the degree to which they deviate from strict consistency in Sections 12.3 and 12.4.

The Web uses caching extensively both at client computers and at proxy servers

maintained by user organizations. The consistency between the copies stored at web

proxies and client caches and the original server is only maintained by explicit user

actions. Clients are not notified when a page stored at the original server is updated; they

must perform explicit checks to keep their local copies up-to-date. This serves the

purposes of web browsing adequately, but it does not support the development of

cooperative applications such as a shared distributed whiteboard. The consistency

mechanisms used in DSM systems are discussed in depth on the companion web site to

the book [www.cdk5.net]. Persistent object systems vary considerably in their approach

to caching and consistency. The CORBA and Persistent Java schemes maintain single

copies of persistent objects, and remote invocation is required to access them, so the

only consistency issue is between the persistent copy of an object on disk and the active

copy in memory, which is not visible to remote clients. The PerDiS and Khazana

projects that we mentioned above maintain cached replicas of objects and employ quite

elaborate consistency mechanisms to produce forms of consistency similar to those

found in DSM systems.

Having introduced some wider issues relating to storage and distribution of

persistent and non-persistent data, we now return to the main topic of this chapter – the

design of basic distributed file systems. We describe some relevant characteristics of

(non-distributed) file systems in Section 12.1.1 and the requirements for distributed file

systems in Section 12.1.2. Section 12.1.3 introduces the case studies that will be used

throughout the chapter. In Section 12.2, we define an abstract model for a basic

Figure 12.2 File system modules

Directory module: relates file names to file IDs

File module: relates file IDs to particular files

Access control module: checks permission for operation requested

File access module: reads or writes file data or attributes

Block module: accesses and allocates disk blocks

Device module: performs disk I/O and bufferingSECTION 12.1 INTRODUCTION 541

distributed file service, including a set of programming interfaces. Sun NFS is described

in Section 12.3; it shares many of the features of the abstract model. In Section 12.4 we

describe the Andrew File System, a widely used system that employs substantially

different caching and consistency mechanisms. Section 12.5 reviews some recent

developments in the design of file services.

The systems described in this chapter do not cover the full spectrum of distributed

file and data management systems. Several systems with more advanced characteristics

will be described later in the book. Chapter 18 includes a description of Coda, a

distributed file system that maintains persistent replicas of files for reliability,

availability and disconnected working. Bayou, a distributed data management system

that provides a weakly consistent form of replication for high availability, is also

covered in Chapter 18. Chapter 20 covers the Tiger video file server, which is designed

to provide timely delivery of streams of data to large numbers of clients. Chapter 21

describes the Google File System (GFS), a file system designed specifically to support

large-scale, data-intensive applications including Internet search.

12.1.1 Characteristics of file systems

File systems are responsible for the organization, storage, retrieval, naming, sharing and

protection of files. They provide a programming interface that characterizes the file

abstraction, freeing programmers from concern with the details of storage allocation and

layout. Files are stored on disks or other non-volatile storage media.

Files contain both data and attributes. The data consist of a sequence of data items

(typically 8-bit bytes), accessible by operations to read and write any portion of the

sequence. The attributes are held as a single record containing information such as the

length of the file, timestamps, file type, owner’s identity and access control lists. A

typical attribute record structure is illustrated in Figure 12.3. The shaded attributes are

managed by the file system and are not normally updatable by user programs.

Figure 12.3 File attribute record structure

File length

Creation timestamp

Read timestamp

Write timestamp

Attribute timestamp

Reference count

Owner

File type

Access control list542 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

File systems are designed to store and manage large numbers of files, with

facilities for creating, naming and deleting files. The naming of files is supported by the

use of directories. A directory is a file, often of a special type, that provides a mapping

from text names to internal file identifiers. Directories may include the names of other

directories, leading to the familiar hierarchic file-naming scheme and the multi-part

pathnames for files used in UNIX and other operating systems. File systems also take

responsibility for the control of access to files, restricting access to files according to

users’ authorizations and the type of access requested (reading, updating, executing and

so on).

The term metadata is often used to refer to all of the extra information stored by

a file system that is needed for the management of files. It includes file attributes,

directories and all the other persistent information used by the file system.

Figure 12.2 shows a typical layered module structure for the implementation of a

non-distributed file system in a conventional operating system. Each layer depends only

on the layers below it. The implementation of a distributed file service requires all of the

components shown there, with additional components to deal with client-server

communication and with the distributed naming and location of files.

File system operations • Figure 12.4 summarizes the main operations on files that are

available to applications in UNIX systems. These are the system calls implemented by

the kernel; application programmers usually access them through procedure libraries

such as the C Standard Input/Output Library or the Java file classes. We give the

primitives here as an indication of the operations that file services are expected to

support and for comparison with the file service interfaces that we shall introduce below.

Figure 12.4 UNIX file system operations

filedes = open(name, mode)

filedes = creat(name, mode)

Opens an existing file with the given name.

Creates a new file with the given name.

Both operations deliver a file descriptor referencing the open

file. The mode is read, write or both.

status = close(filedes) Closes the open file filedes.

count = read(filedes, buffer, n)

count = write(filedes, buffer, n)

Transfers n bytes from the file referenced by filedes to buffer.

Transfers n bytes to the file referenced by filedes from buffer.

Both operations deliver the number of bytes actually

transferred and advance the read-write pointer.

pos = lseek(filedes, offset,

whence)

Moves the read-write pointer to offset (relative or absolute,

depending on whence).

status = unlink(name) Removes the file name from the directory structure. If the file

has no other names, it is deleted.

status = link(name1, name2) Adds a new name (name2) for a file (name1).

status = stat(name, buffer) Puts the file attributes for file name into buffer.SECTION 12.1 INTRODUCTION 543

The UNIX operations are based on a programming model in which some file state

information is stored by the file system for each running program. This consists of a list

of currently open files with a read-write pointer for each, giving the position within the

file at which the next read or write operation will be applied.

The file system is responsible for applying access control for files. In local file

systems such as UNIX, it does so when each file is opened, checking the rights allowed

for the user’s identity in the access control list against the mode of access requested in

the open system call. If the rights match the mode, the file is opened and the mode is

recorded in the open file state information.

12.1.2 Distributed file system requirements

Many of the requirements and potential pitfalls in the design of distributed services were

first observed in the early development of distributed file systems. Initially, they offered

access transparency and location transparency; performance, scalability, concurrency

control, fault tolerance and security requirements emerged and were met in subsequent

phases of development. We discuss these and related requirements in the following

subsections.

Transparency • The file service is usually the most heavily loaded service in an intranet,

so its functionality and performance are critical. The design of the file service should

support many of the transparency requirements for distributed systems identified in

Section 1.5.7. The design must balance the flexibility and scalability that derive from

transparency against software complexity and performance. The following forms of

transparency are partially or wholly addressed by current file services:

Access transparency: Client programs should be unaware of the distribution of files.

A single set of operations is provided for access to local and remote files. Programs

written to operate on local files are able to access remote files without modification.

Location transparency: Client programs should see a uniform file name space. Files

or groups of files may be relocated without changing their pathnames, and user

programs see the same name space wherever they are executed.

Mobility transparency: Neither client programs nor system administration tables in

client nodes need to be changed when files are moved. This allows file mobility –

files or, more commonly, sets or volumes of files may be moved, either by system

administrators or automatically.

Performance transparency: Client programs should continue to perform satisfactorily while the load on the service varies within a specified range.

Scaling transparency: The service can be expanded by incremental growth to deal

with a wide range of loads and network sizes.

Concurrent file updates • Changes to a file by one client should not interfere with the

operation of other clients simultaneously accessing or changing the same file. This is the

well-known issue of concurrency control, discussed in detail in Chapter 16. The need for

concurrency control for access to shared data in many applications is widely accepted

and techniques are known for its implementation, but they are costly. Most current file544 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

services follow modern UNIX standards in providing advisory or mandatory file- or

record-level locking.

File replication • In a file service that supports replication, a file may be represented by

several copies of its contents at different locations. This has two benefits – it enables

multiple servers to share the load of providing a service to clients accessing the same set

of files, enhancing the scalability of the service, and it enhances fault tolerance by

enabling clients to locate another server that holds a copy of the file when one has failed.

Few file services support replication fully, but most support the caching of files or

portions of files locally, a limited form of replication. The replication of data is

discussed in Chapter 18, which includes a description of the Coda replicated file service.

Hardware and operating system heterogeneity • The service interfaces should be defined so that client and server software can be implemented for different operating systems and computers. This requirement is an important aspect of openness.

Fault tolerance • The central role of the file service in distributed systems makes it

essential that the service continue to operate in the face of client and server failures.

Fortunately, a moderately fault-tolerant design is straightforward for simple servers. To

cope with transient communication failures, the design can be based on at-most-once

invocation semantics (see Section 5.3.1); or it can use the simpler at-least-once

semantics with a server protocol designed in terms of idempotent operations, ensuring

that duplicated requests do not result in invalid updates to files. The servers can be

stateless, so that they can be restarted and the service restored after a failure without any

need to recover previous state. Tolerance of disconnection or server failures requires file

replication, which is more difficult to achieve and will be discussed in Chapter 18.

Consistency • Conventional file systems such as that provided in UNIX offer one-copy

update semantics. This refers to a model for concurrent access to files in which the file

contents seen by all of the processes accessing or updating a given file are those that they

would see if only a single copy of the file contents existed. When files are replicated or

cached at different sites, there is an inevitable delay in the propagation of modifications

made at one site to all of the other sites that hold copies, and this may result in some

deviation from one-copy semantics.

Security • Virtually all file systems provide access-control mechanisms based on the

use of access control lists. In distributed file systems, there is a need to authenticate

client requests so that access control at the server is based on correct user identities and

to protect the contents of request and reply messages with digital signatures and

(optionally) encryption of secret data. We discuss the impact of these requirements in

the case studies later in this chapter.

Efficiency • A distributed file service should offer facilities that are of at least the same

power and generality as those found in conventional file systems and should achieve a

comparable level of performance. Birrell and Needham [1980] expressed their design

aims for the Cambridge File Server (CFS) in these terms:

We would wish to have a simple, low-level file server in order to share an

expensive resource, namely a disk, whilst leaving us free to design the filing

system most appropriate to a particular client, but we would wish also to have

available a high-level system shared between clients.SECTION 12.1 INTRODUCTION 545

The changed economics of disk storage have reduced the significance of their first goal,

but their perception of the need for a range of services addressing the requirements of

clients with different goals remains and can best be addressed by a modular architecture

of the type outlined above.

The techniques used for the implementation of file services are an important part

of the design of distributed systems. A distributed file system should provide a service

that is comparable with, or better than, local file systems in performance and reliability.

It must be convenient to administer, providing operations and tools that enable system

administrators to install and operate the system conveniently.

12.1.3 Case studies

We have constructed an abstract model for a file service to act as an introductory

example, separating implementation concerns and providing a simplified model. We

describe the Sun Network File System in some detail, drawing on our simpler abstract

model to clarify its architecture. The Andrew File System is then described, providing a

view of a distributed file system that takes a different approach to scalability and

consistency maintenance.

File service architecture • This is an abstract architectural model that underpins both

NFS and AFS. It is based upon a division of responsibilities between three modules – a

client module that emulates a conventional file system interface for application

programs, and server modules, that perform operations for clients on directories and on

files. The architecture is designed to enable a stateless implementation of the server

module.

SUN NFS • Sun Microsystems’s Network File System (NFS) has been widely adopted

in industry and in academic environments since its introduction in 1985. The design and

development of NFS were undertaken by staff at Sun Microsystems in 1984 [Sandberg

et al. 1985, Sandberg 1987, Callaghan 1999]. Although several distributed file services

had already been developed and used in universities and research laboratories, NFS was

the first file service that was designed as a product. The design and implementation of

NFS have achieved success both technically and commercially.

To encourage its adoption as a standard, the definitions of the key interfaces were

placed in the public domain [Sun 1989], enabling other vendors to produce

implementations, and the source code for a reference implementation was made

available to other computer vendors under licence. It is now supported by many vendors,

and the NFS protocol (version 3) is an Internet standard, defined in RFC 1813

[Callaghan et al. 1995]. Callaghan’s book on NFS [Callaghan 1999] is an excellent

source on the design and development of NFS and related topics.

NFS provides transparent access to remote files for client programs running on

UNIX and other systems. The client-server relationship is symmetrical: each computer

in an NFS network can act as both a client and a server, and the files at every machine

can be made available for remote access by other machines. Any computer can be a

server, exporting some of its files, and a client, accessing files on other machines. But it

is common practice to configure larger installations with some machines as dedicated

servers and others as workstations.546 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

An important goal of NFS is to achieve a high level of support for hardware and

operating system heterogeneity. The design is operating system–independent: client and

server implementations exist for almost all operating systems and platforms, including

all versions of Windows, Mac OS, Linux and every other version of UNIX.

Implementations of NFS on high-performance multiprocessor hosts have been

developed by several vendors, and these are widely used to meet storage requirements

in intranets with many concurrent users.

Andrew File System • Andrew is a distributed computing environment developed at

Carnegie Mellon University (CMU) for use as a campus computing and information

system [Morris et al. 1986]. The design of the Andrew File System (henceforth

abbreviated AFS) reflects an intention to support information sharing on a large scale by

minimizing client-server communication. This is achieved by transferring whole files

between server and client computers and caching them at clients until the server receives

a more up-to-date version. We shall describe AFS-2, the first ‘production’

implementation, following the descriptions by Satyanarayanan [1989a, 1989b]. More

recent descriptions can be found in Campbell [1997] and [Linux AFS].

AFS was initially implemented on a network of workstations and servers running

BSD UNIX and the Mach operating system at CMU and was subsequently made

available in commercial and public-domain versions. A public-domain implementation

of AFS is available in the Linux operating system [Linux AFS]. AFS was adopted as the

basis for the DCE/DFS file system in the Open Software Foundation’s Distributed

Computing Environment (DCE) [www.opengroup.org]. The design of DCE/DFS went

beyond AFS in several important respects, which we outline in Section 12.5.

12.2 File service architecture

An architecture that offers a clear separation of the main concerns in providing access

to files is obtained by structuring the file service as three components – a flat file service,

a directory service and a client module. The relevant modules and their relationships are

shown in Figure 12.5. The flat file service and the directory service each export an

interface for use by client programs, and their RPC interfaces, taken together, provide a

comprehensive set of operations for access to files. The client module provides a single

programming interface with operations on files similar to those found in conventional

file systems. The design is open in the sense that different client modules can be used to

implement different programming interfaces, simulating the file operations of a variety

of different operating systems and optimizing the performance for different client and

server hardware configurations.

The division of responsibilities between the modules can be defined as follows:

Flat file service • The flat file service is concerned with implementing operations on the

contents of files. Unique file identifiers (UFIDs) are used to refer to files in all requests

for flat file service operations. The division of responsibilities between the file service

and the directory service is based upon the use of UFIDs. UFIDs are long sequences of

bits chosen so that each file has a UFID that is unique among all of the files in aSECTION 12.2 FILE SERVICE ARCHITECTURE 547

distributed system. When the flat file service receives a request to create a file, it

generates a new UFID for it and returns the UFID to the requester.

Directory service • The directory service provides a mapping between text names for

files and their UFIDs. Clients may obtain the UFID of a file by quoting its text name to

the directory service. The directory service provides the functions needed to generate

directories, to add new file names to directories and to obtain UFIDs from directories. It

is a client of the flat file service; its directory files are stored in files of the flat file

service. When a hierarchic file-naming scheme is adopted, as in UNIX, directories hold

references to other directories.

Client module • A client module runs in each client computer, integrating and

extending the operations of the flat file service and the directory service under a single

application programming interface that is available to user-level programs in client

computers. For example, in UNIX hosts, a client module would be provided that

emulates the full set of UNIX file operations, interpreting UNIX multi-part file names

by iterative requests to the directory service. The client module also holds information

about the network locations of the flat file server and directory server processes. Finally,

the client module can play an important role in achieving satisfactory performance

through the implementation of a cache of recently used file blocks at the client.

Flat file service interface • Figure 12.6 contains a definition of the interface to a flat file

service. This is the RPC interface used by client modules. It is not normally used directly

by user-level programs. A FileId is invalid if the file that it refers to is not present in the

server processing the request or if its access permissions are inappropriate for the

operation requested. All of the procedures in the interface except Create throw

exceptions if the FileId argument contains an invalid UFID or the user doesn’t have

sufficient access rights. These exceptions are omitted from the definition for clarity.

The most important operations are those for reading and writing. Both the Read

and the Write operation require a parameter i specifying a position in the file. The Read

operation copies the sequence of n data items beginning at item i from the specified file

Figure 12.5 File service architecture

Client computer Server computer

Application

program

Application

program

Client module

Flat file service

Directory service548 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

into Data, which is then returned to the client. The Write operation copies the sequence

of data items in Data into the specified file beginning at item i, replacing the previous

contents of the file at the corresponding position and extending the file if necessary.

Create creates a new, empty file and returns the UFID that is generated. Delete

removes the specified file.

GetAttributes and SetAttributes enable clients to access the attribute record.

GetAttributes is normally available to any client that is allowed to read the file. Access

to the SetAttributes operation would normally be restricted to the directory service that

provides access to the file. The values of the length and timestamp portions of the

attribute record are not affected by SetAttributes; they are maintained separately by the

flat file service itself.

Comparison with UNIX: Our interface and the UNIX file system primitives are

functionally equivalent. It is a simple matter to construct a client module that emulates

the UNIX system calls in terms of our flat file service and the directory service

operations described in the next section.

In comparison with the UNIX interface, our flat file service has no open and close

operations – files can be accessed immediately by quoting the appropriate UFID. The

Read and Write requests in our interface include a parameter specifying a starting point

within the file for each transfer, whereas the equivalent UNIX operations do not. In

UNIX, each read or write operation starts at the current position of the read-write

pointer, and the read-write pointer is advanced by the number of bytes transferred after

each read or write. A seek operation is provided to enable the read-write pointer to be

explicitly repositioned.

The interface to our flat file service differs from the UNIX file system interface

mainly for reasons of fault tolerance:

Repeatable operations: With the exception of Create, the operations are

idempotent, allowing the use of at-least-once RPC semantics – clients may repeat

calls to which they receive no reply. Repeated execution of Create produces a

different new file for each call.

Figure 12.6 Flat file service operations

Read(FileId, i, n) → Data

— throws BadPosition

If 1 ≤ i ≤ Length(File): Reads a sequence of up to n items

from a file starting at item i and returns it in Data.

Write(FileId, i, Data)

— throws BadPosition

If 1 ≤ i ≤ Length(File)+1: Writes a sequence of Data to a

file, starting at item i, extending the file if necessary.

Create() → FileId Creates a new file of length 0 and delivers a UFID for it.

Delete(FileId) Removes the file from the file store.

GetAttributes(FileId) → Attr Returns the file attributes for the file.

SetAttributes(FileId, Attr) Sets the file attributes (only those attributes that are not

shaded in Figure 12.3).SECTION 12.2 FILE SERVICE ARCHITECTURE 549

Stateless servers: The interface is suitable for implementation by stateless servers.

Stateless servers can be restarted after a failure and resume operation without any

need for clients or the server to restore any state.

The UNIX file operations are neither idempotent nor consistent with the requirement for

a stateless implementation. A read-write pointer is generated by the UNIX file system

whenever a file is opened, and it is retained, together with the results of access-control

checks, until the file is closed. The UNIX read and write operations are not idempotent;

if an operation is accidentally repeated, the automatic advance of the read-write pointer

results in access to a different portion of the file in the repeated operation. The read-write

pointer is a hidden, client-related state variable. To mimic it in a file server, open and

close operations would be needed, and the read-write pointer’s value would have to be

retained by the server as long as the relevant file is open. By eliminating the read-write

pointer, we have eliminated most of the need for the file server to retain state

information on behalf of specific clients.

Access control • In the UNIX file system, the user’s access rights are checked against

the access mode (read or write) requested in the open call (Figure 12.4 shows the UNIX

file system API) and the file is opened only if the user has the necessary rights. The user

identity (UID) used in the access rights check is retrieved during the user’s earlier

authenticated login and cannot be tampered with in non-distributed implementations.

The resulting access rights are retained until the file is closed, and no further checks are

required when subsequent operations on the same file are requested.

In distributed implementations, access rights checks have to be performed at the

server because the server RPC interface is an otherwise unprotected point of access to

files. A user identity has to be passed with requests, and the server is vulnerable to

forged identities. Furthermore, if the results of an access rights check were retained at

the server and used for future accesses, the server would no longer be stateless. Two

alternative approaches to the latter problem can be adopted:

• An access check is made whenever a file name is converted to a UFID, and the

results are encoded in the form of a capability (see Section 11.2.4), which is

returned to the client for submission with subsequent requests.

• A user identity is submitted with every client request, and access checks are

performed by the server for every file operation.

Both methods enable stateless server implementation, and both have been used in

distributed file systems. The second is more common; it is used in both NFS and AFS.

Neither of these approaches overcomes the security problem concerning forged user

identities, but we saw in Chapter 11 that this can be addressed by the use of digital

signatures. Kerberos is an effective authentication scheme that has been applied to both

NFS and AFS.

In our abstract model, we make no assumption about the method by which access

control is implemented. The user identity is passed as an implicit parameter and can be

used whenever it is needed.

Directory service interface • Figure 12.7 contains a definition of the RPC interface to a

directory service. The primary purpose of the directory service is to provide a service for

translating text names to UFIDs. In order to do so, it maintains directory files containing550 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

the mappings between text names for files and UFIDs. Each directory is stored as a

conventional file with a UFID, so the directory service is a client of the file service.

We define only operations on individual directories. For each operation, a UFID

for the file containing the directory is required (in the Dir parameter). The Lookup

operation in the basic directory service performs a single Name → UFID translation. It

is a building block for use in other services or in the client module to perform more

complex translations, such as the hierarchic name interpretation found in UNIX. As

before, exceptions caused by inadequate access rights are omitted from the definitions.

There are two operations for altering directories: AddName and UnName.

AddName adds an entry to a directory and increments the reference count field in the

file’s attribute record.

UnName removes an entry from a directory and decrements the reference count.

If this causes the reference count to reach zero, the file is removed. GetNames is

provided to enable clients to examine the contents of directories and to implement

pattern-matching operations on file names such as those found in the UNIX shell. It

returns all or a subset of the names stored in a given directory. The names are selected

by pattern matching against a regular expression supplied by the client.

The provision of pattern matching in the GetNames operation enables users to

determine the names of one or more files by giving an incomplete specification of the

characters in the names. A regular expression is a specification for a class of strings in

the form of an expression containing a combination of literal substrings and symbols

denoting variable characters or repeated occurrences of characters or substrings.

Hierarchic file system • A hierarchic file system such as the one that UNIX provides

consists of a number of directories arranged in a tree structure. Each directory holds the

names of the files and other directories that are accessible from it. Any file or directory

can be referenced using a pathname – a multi-part name that represents a path through

Figure 12.7 Directory service operations

Lookup(Dir, Name) → FileId

— throws NotFound

Locates the text name in the directory and returns the

relevant UFID. If Name is not in the directory, throws an

exception.

AddName(Dir, Name, FileId)

— throws NameDuplicate

If Name is not in the directory, adds (Name, File) to the

directory and updates the file’s attribute record.

If Name is already in the directory, throws an exception.

UnName(Dir, Name)

— throws NotFound

If Name is in the directory, removes the entry containing

Name from the directory.

If Name is not in the directory, throws an exception.

GetNames(Dir, Pattern) → NameSeq Returns all the text names in the directory that match the

regular expression Pattern.SECTION 12.2 FILE SERVICE ARCHITECTURE 551

the tree. The root has a distinguished name, and each file or directory has a name in a

directory. The UNIX file-naming scheme is not a strict hierarchy – files can have several

names, and they can be in the same or different directories. This is implemented by a

link operation, which adds a new name for a file to a specified directory.

A UNIX-like file-naming system can be implemented by the client module using

the flat file and directory services that we have defined. A tree-structured network of

directories is constructed with files at the leaves and directories at the other nodes of the

tree. The root of the tree is a directory with a ‘well-known’ UFID. Multiple names for

files can be supported using the AddName operation and the reference count field in the

attribute record.

A function can be provided in the client module that gets the UFID of a file given

its pathname. The function interprets the pathname starting from the root, using Lookup

to obtain the UFID of each directory in the path.

In a hierarchic directory service, the file attributes associated with files should

include a type field that distinguishes between ordinary files and directories. This is used

when following a path to ensure that each part of the name, except the last, refers to a

directory.

File groups • A file group is a collection of files located on a given server. A server may

hold several file groups, and groups can be moved between servers, but a file cannot

change the group to which it belongs. A similar construct called a filesystem is used in

UNIX and in most other operating systems. (Terminology note: the single word

filesystem refers to the set of files held in a storage device or partition, whereas the words

file system refer to a software component that provides access to files.) File groups were

originally introduced to support facilities for moving collections of files stored on

removable media between computers. In a distributed file service, file groups support

the allocation of files to file servers in larger logical units and enable the service to be

implemented with files stored on several servers. In a distributed file system that

supports file groups, the representation of UFIDs includes a file group identifier

component, enabling the client module in each client computer to take responsibility for

dispatching requests to the server that holds the relevant file group.

File group identifiers must be unique throughout a distributed system. Since file

groups can be moved and distributed systems that are initially separate can be merged

to form a single system, the only way to ensure that file group identifiers will always be

distinct in a given system is to generate them with an algorithm that ensures global

uniqueness. For example, whenever a new file group is created, a unique identifier can

be generated by concatenating the 32-bit IP address of the host creating the new group

with a 16-bit integer derived from the date, producing a unique 48-bit integer:

Note that the IP address cannot be used for the purpose of locating the file group, since

it may be moved to another server. Instead, a mapping between group identifiers and

servers should be maintained by the file service.

32 bits 16 bits

file group identifier: IP address date552 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

12.3 Case study: Sun Network File System

Figure 12.8 shows the architecture of Sun NFS. It follows the abstract model defined in

the preceding section. All implementations of NFS support the NFS protocol – a set of

remote procedure calls that provide the means for clients to perform operations on a

remote file store. The NFS protocol is operating system–independent but was originally

developed for use in networks of UNIX systems, and we shall describe the UNIX

implementation the NFS protocol (version 3).

The NFS server module resides in the kernel on each computer that acts as an NFS

server. Requests referring to files in a remote file system are translated by the client

module to NFS protocol operations and then passed to the NFS server module at the

computer holding the relevant file system.

The NFS client and server modules communicate using remote procedure calls.

Sun’s RPC system, described in Section 5.3.3, was developed for use in NFS. It can be

configured to use either UDP or TCP, and the NFS protocol is compatible with both. A

port mapper service is included to enable clients to bind to services in a given host by

name. The RPC interface to the NFS server is open: any process can send requests to an

NFS server; if the requests are valid and they include valid user credentials, they will be

acted upon. The submission of signed user credentials can be required as an optional

security feature, as can the encryption of data for privacy and integrity.

Virtual file system • Figure 12.8 makes it clear that NFS provides access transparency:

user programs can issue file operations for local or remote files without distinction.

Other distributed file systems may be present that support UNIX system calls, and if so,

they could be integrated in the same way.

Figure 12.8 NFS architecture

UNIX kernel

protocol

Client computer Server computer

system calls

Local Remote

UNIX

file

system

NFS

client

NFS

server

UNIX

file

system

Application

program

Application

program

NFS

UNIX

UNIX kernel

Virtual file system

Other

file system

Virtual file systemSECTION 12.3 CASE STUDY: SUN NETWORK FILE SYSTEM 553

The integration is achieved by a virtual file system (VFS) module, which has been

added to the UNIX kernel to distinguish between local and remote files and to translate

between the UNIX-independent file identifiers used by NFS and the internal file

identifiers normally used in UNIX and other file systems. In addition, VFS keeps track

of the filesystems that are currently available both locally and remotely, and it passes

each request to the appropriate local system module (the UNIX file system, the NFS

client module or the service module for another file system).

The file identifiers used in NFS are called file handles. A file handle is opaque to

clients and contains whatever information the server needs to distinguish an individual

file. In UNIX implementations of NFS, the file handle is derived from the file’s i-node

number by adding two extra fields as follows (the i-node number of a UNIX file is a

number that serves to identify and locate the file within the file system in which the file

is stored):

NFS adopts the UNIX mountable filesystem as the unit of file grouping defined in the

preceding section. The filesystem identifier field is a unique number that is allocated to

each filesystem when it is created (and in the UNIX implementation is stored in the

superblock of the file system). The i-node generation number is needed because in the

conventional UNIX file system i-node numbers are reused after a file is removed. In the

VFS extensions to the UNIX file system, a generation number is stored with each file

and is incremented each time the i-node number is reused (for example, in a UNIX creat

system call). The client obtains the first file handle for a remote file system when it

mounts it. File handles are passed from server to client in the results of lookup, create

and mkdir operations (see Figure 12.9) and from client to server in the argument lists of

all server operations.

The virtual file system layer has one VFS structure for each mounted file system

and one v-node per open file. A VFS structure relates a remote file system to the local

directory on which it is mounted. The v-node contains an indicator to show whether a

file is local or remote. If the file is local, the v-node contains a reference to the index of

the local file (an i-node in a UNIX implementation). If the file is remote, it contains the

file handle of the remote file.

Client integration • The NFS client module plays the role described for the client

module in our architectural model, supplying an interface suitable for use by

conventional application programs. But unlike our model client module, it emulates the

semantics of the standard UNIX file system primitives precisely and is integrated with

the UNIX kernel. It is integrated with the kernel and not supplied as a library for loading

into client processes so that:

• user programs can access files via UNIX system calls without recompilation or

reloading;

• a single client module serves all of the user-level processes, with a shared cache

of recently used blocks (described below);

File handle: Filesystem identifier i-node number

of file

i-node generation

number554 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

• the encryption key used to authenticate user IDs passed to the server (see below)

can be retained in the kernel, preventing impersonation by user-level clients.

The NFS client module cooperates with the virtual file system in each client machine. It

operates in a similar manner to the conventional UNIX file system, transferring blocks

of files to and from the server and caching the blocks in the local memory whenever

possible. It shares the same buffer cache that is used by the local input-output system.

Figure 12.9 NFS server operations (NFS version 3 protocol, simplified)

lookup(dirfh, name) → fh, attr Returns file handle and attributes for the file name in the directory dirfh.

create(dirfh, name, attr) →

newfh, attr

Creates a new file name in directory dirfh with attributes attr and returns

the new file handle and attributes.

remove(dirfh, name) → status Removes file name from directory dirfh.

getattr(fh) → attr Returns file attributes of file fh. (Similar to the UNIX stat system call.)

setattr(fh, attr) → attr Sets the attributes (mode, user ID, group ID, size, access time and

modify time of a file). Setting the size to 0 truncates the file.

read(fh, offset, count) → attr, data Returns up to count bytes of data from a file starting at offset. Also

returns the latest attributes of the file.

write(fh, offset, count, data) → attr Writes count bytes of data to a file starting at offset. Returns the

attributes of the file after the write has taken place.

rename(dirfh, name, todirfh,

toname) → status

Changes the name of file name in directory dirfh to toname in directory

todirfh.

link(newdirfh, newname, fh)

→ status

Creates an entry newname in the directory newdirfh that refers to the file

or directory fh.

symlink(newdirfh, newname, string)

→ status

Creates an entry newname in the directory newdirfh of type symbolic

link with the value string. The server does not interpret the string but

makes a symbolic link file to hold it.

readlink(fh) → string Returns the string that is associated with the symbolic link file identified

by fh.

mkdir(dirfh, name, attr) → newfh,

attr

Creates a new directory name with attributes attr and returns the new

file handle and attributes.

rmdir(dirfh, name) → status Removes the empty directory name from the parent directory dirfh.

Fails if the directory is not empty.

readdir(dirfh, cookie, count) →

entries

Returns up to count bytes of directory entries from the directory dirfh.

Each entry contains a file name, a file handle and an opaque pointer to

the next directory entry, called a cookie. The cookie is used in

subsequent readdir calls to start reading from the following entry. If the

value of cookie is 0, reads from the first entry in the directory.

statfs(fh) → fsstats Returns file system information (such as block size, number of free

blocks and so on) for the file system containing a file fh.SECTION 12.3 CASE STUDY: SUN NETWORK FILE SYSTEM 555

But since several clients in different host machines may simultaneously access the same

remote file, a new and significant cache consistency problem arises.

Access control and authentication • Unlike the conventional UNIX file system, the NFS

server is stateless and does not keep files open on behalf of its clients. So the server must

check the user’s identity against the file’s access permission attributes afresh on each

request, to see whether the user is permitted to access the file in the manner requested.

The Sun RPC protocol requires clients to send user authentication information (for

example, the conventional UNIX 16-bit user ID and group ID) with each request and this

is checked against the access permission in the file attributes. These additional

parameters are not shown in our overview of the NFS protocol in Figure 12.9; they are

supplied automatically by the RPC system.

In its simplest form, there is a security loophole in this access-control mechanism.

An NFS server provides a conventional RPC interface at a well-known port on each host

and any process can behave as a client, sending requests to the server to access or update

a file. The client can modify the RPC calls to include the user ID of any user,

impersonating the user without their knowledge or permission. This security loophole

has been closed by the use of an option in the RPC protocol for the DES encryption of

the user’s authentication information. More recently, Kerberos has been integrated with

Sun NFS to provide a stronger and more comprehensive solution to the problems of user

authentication and security; we describe this below.

NFS server interface • A simplified representation of the RPC interface provided by

NFS version 3 servers (defined in RFC 1813 [Callaghan et al. 1995]) is shown in Figure

12.9. The NFS file access operations read, write, getattr and setattr are almost identical

to the Read, Write, GetAttributes and SetAttributes operations defined for our flat file

service model (Figure 12.6). The lookup operation and most of the other directory

operations defined in Figure 12.9 are similar to those in our directory service model

(Figure 12.7).

The file and directory operations are integrated in a single service; the creation and

insertion of file names in directories is performed by a single create operation, which

takes the text name of the new file and the file handle for the target directory as

arguments. The other NFS operations on directories are create, remove, rename, link,

symlink, readlink, mkdir, rmdir, readdir and statfs. They resemble their UNIX

counterparts with the exception of readdir, which provides a representationindependent method for reading the contents of directories, and statfs, which gives the

status information on remote file systems.

Mount service • The mounting of subtrees of remote filesystems by clients is supported

by a separate mount service process that runs at user level on each NFS server computer.

On each server, there is a file with a well-known name (/etc/exports) containing the

names of local filesystems that are available for remote mounting. An access list is

associated with each filesystem name indicating which hosts are permitted to mount the

filesystem.

Clients use a modified version of the UNIX mount command to request mounting

of a remote filesystem, specifying the remote host’s name, the pathname of a directory

in the remote filesystem and the local name with which it is to be mounted. The remote

directory may be any subtree of the required remote filesystem, enabling clients to

mount any part of the remote filesystem. The modified mount command communicates556 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

with the mount service process on the remote host using a mount protocol. This is an

RPC protocol and includes an operation that takes a directory pathname and returns the

file handle of the specified directory if the client has access permission for the relevant

filesystem. The location (IP address and port number) of the server and the file handle

for the remote directory are passed on to the VFS layer and the NFS client.

Figure 12.10 illustrates a Client with two remotely mounted file stores. The nodes

people and users in filesystems at Server 1 and Server 2 are mounted over nodes students

and staff in Client’s local file store. The meaning of this is that programs running at

Client can access files at Server 1 and Server 2 by using pathnames such as

/usr/students/jon and /usr/staff/ann.

Remote filesystems may be hard-mounted or soft-mounted in a client computer.

When a user-level process accesses a file in a filesystem that is hard-mounted, the

process is suspended until the request can be completed, and if the remote host is

unavailable for any reason the NFS client module continues to retry the request until it

is satisfied. Thus in the case of a server failure, user-level processes are suspended until

the server restarts and then they continue just as though there had been no failure. But if

the relevant filesystem is soft-mounted, the NFS client module returns a failure

indication to user-level processes after a small number of retries. Properly constructed

programs will then detect the failure and take appropriate recovery or reporting actions.

But many UNIX utilities and applications do not test for the failure of file access

operations, and these behave in unpredictable ways in the case of failure of a softmounted filesystem. For this reason, many installations use hard mounting exclusively,

with the consequence that programs are unable to recover gracefully when an NFS

server is unavailable for a significant period.

Pathname translation • UNIX file systems translate multi-part file pathnames to i-node

references in a step-by-step process whenever the open, creat or stat system calls are

used. In NFS, pathnames cannot be translated at a server, because the name may cross a

Figure 12.10 Local and remote filesystems accessible on an NFS client

jim jane joe ann

students users

vmunix usr

Client Server 2

. . . nfs

Remote

staff mount

big bob jon

people

Server 1

export

(root)

Remote

mount

. . .

x

Note: The file system mounted at /usr/students in the client is actually the subtree located

at /export/people in Server 1; the filesystem mounted at /usr/staff in the client is

actually the subtree located at /nfs/users in Server 2.

(root) (root)SECTION 12.3 CASE STUDY: SUN NETWORK FILE SYSTEM 557

‘mount point’ at the client – directories holding different parts of a multi-part name may

reside in filesystems at different servers. So pathnames are parsed, and their translation

is performed in an iterative manner by the client. Each part of a name that refers to a

remote-mounted directory is translated to a file handle using a separate lookup request

to the remote server.

The lookup operation looks for a single part of a pathname in a given directory and

returns the corresponding file handle and file attributes. The file handle returned in the

previous step is used as a parameter in the next lookup step. Since file handles are

opaque to NFS client code, the virtual file system is responsible for resolving file

handles to a local or a remote directory and performing the necessary indirection when

it references a local mount point. Caching of the results of each step in pathname

translations alleviates the apparent inefficiency of this process, taking advantage of

locality of reference to files and directories; users and programs typically access files in

only one or a small number of directories.

Automounter • The automounter was added to the UNIX implementation of NFS in

order to mount a remote directory dynamically whenever an ‘empty’ mount point is

referenced by a client. The original implementation of the automounter ran as a userlevel UNIX process in each client computer. Later versions (called autofs) were

implemented in the kernel for Solaris and Linux. We describe the original version here.

The automounter maintains a table of mount points (pathnames) with a reference

to one or more NFS servers listed against each. It behaves like a local NFS server at the

client machine. When the NFS client module attempts to resolve a pathname that

includes one of these mount points, it passes to the local automounter a lookup() request

that locates the required filesystem in its table and sends a ‘probe’ request to each server

listed. The filesystem on the first server to respond is then mounted at the client using

the normal mount service. The mounted filesystem is linked to the mount point using a

symbolic link, so that accesses to it will not result in further requests to the automounter.

File access then proceeds in the normal way without further reference to the

automounter unless there are no references to the symbolic link for several minutes. In

the latter case, the automounter unmounts the remote filesystem.

The later kernel implementations replaced the symbolic links with real mounts,

avoiding some problems that arose with applications that cached the temporary

pathnames used in user-level automounters [Callaghan 1999].

A simple form of read-only replication can be achieved by listing several servers

containing identical copies of a filesystem or file subtree against a name in the

automounter table. This is useful for heavily used file systems that change infrequently,

such as UNIX system binaries. For example, copies of the /usr/lib directory and its

subtree might be held on more than one server. On the first occasion that a file in /usr/lib

is opened at a client, all of the servers will be sent probe messages, and the first to

respond will be mounted at the client. This provides a limited degree of fault tolerance

and load balancing, since the first server to respond will be one that has not failed and is

likely to be one that is not heavily occupied with servicing other requests.

Server caching • Caching in both the client and the server computer are indispensable

features of NFS implementations in order to achieve adequate performance.

In conventional UNIX systems, file pages, directories and file attributes that have

been read from disk are retained in a main memory buffer cache until the buffer space558 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

is required for other pages. If a process then issues a read or a write request for a page

that is already in the cache, it can be satisfied without another disk access. Read-ahead

anticipates read accesses and fetches the pages following those that have most recently

been read, and delayed-write optimizes writes: when a page has been altered (by a write

request), its new contents are written to disk only when the buffer page is required for

another page. To guard against loss of data in a system crash, the UNIX sync operation

flushes altered pages to disk every 30 seconds. These caching techniques work in a

conventional UNIX environment because all read and write requests issued by userlevel processes pass through a single cache that is implemented in the UNIX kernel

space. The cache is always kept up-to-date, and file accesses cannot bypass the cache.

NFS servers use the cache at the server machine just as it is used for other file

accesses. The use of the server’s cache to hold recently read disk blocks does not raise

any consistency problems; but when a server performs write operations, extra measures

are needed to ensure that clients can be confident that the results of the write operations

are persistent, even when server crashes occur. In version 3 of the NFS protocol, the

write operation offers two options for this (not shown in Figure 12.9):

1. Data in write operations received from clients is stored in the memory cache at the

server and written to disk before a reply is sent to the client. This is called writethrough caching. The client can be sure that the data is stored persistently as soon

as the reply has been received.

2. Data in write operations is stored only in the memory cache. It will be written to

disk when a commit operation is received for the relevant file. The client can be

sure that the data is persistently stored only when a reply to a commit operation for

the relevant file has been received. Standard NFS clients use this mode of

operation, issuing a commit whenever a file that was open for writing is closed.

Commit is an additional operation provided in version 3 of the NFS protocol; it was

added to overcome a performance bottleneck caused by the write-through mode of

operation in servers that receive large numbers of write operations.

The requirement for write-through in distributed file systems is an instance of the

independent failure modes discussed in Chapter 1 – clients continue to operate when a

server fails, and application programs may take actions on the assumption that the

results of previous writes are committed to disk storage. This is unlikely to occur in the

case of local file updates, because the failure of a local file system is almost certain to

result in the failure of all the application processes running on the same computer.

Client caching • The NFS client module caches the results of read, write, getattr,

lookup and readdir operations in order to reduce the number of requests transmitted to

servers. Client caching introduces the potential for different versions of files or portions

of files to exist in different client nodes, because writes by a client do not result in the

immediate updating of cached copies of the same file in other clients. Instead, clients are

responsible for polling the server to check the currency of the cached data that they hold.

A timestamp-based method is used to validate cached blocks before they are used.

Each data or metadata item in the cache is tagged with two timestamps:

Tc is the time when the cache entry was last validated.

Tm is the time when the block was last modified at the server.SECTION 12.3 CASE STUDY: SUN NETWORK FILE SYSTEM 559

A cache entry is valid at time T if T – Tc is less than a freshness interval t, or if the value

for Tm recorded at the client matches the value of Tm at the server (that is, the data has

not been modified at the server since the cache entry was made). Formally, the validity

condition is:

The selection of a value for t involves a compromise between consistency and efficiency.

A very short freshness interval will result in a close approximation to one-copy

consistency, at the cost of a relatively heavy load of calls to the server to check the value

of Tmserver. In Sun Solaris clients, t is set adaptively for individual files to a value in the

range 3 to 30 seconds, depending on the frequency of updates to the file. For directories

the range is 30 to 60 seconds, reflecting the lower risk of concurrent updates.

There is one value of Tmserver for all the data blocks in a file and another for the

file attributes. Since NFS clients cannot determine whether a file is being shared or not,

the validation procedure must be used for all file accesses. A validity check is performed

whenever a cache entry is used. The first half of the validity condition can be evaluated

without access to the server. If it is true, then the second half need not be evaluated; if it

is false, the current value of Tmserver is obtained (by means of a getattr call to the server)

and compared with the local value Tmclient. If they are the same, then the cache entry is

taken to be valid and the value of Tc for that cache entry is updated to the current time.

If they differ, then the cached data has been updated at the server and the cache entry is

invalidated, resulting in a request to the server for the relevant data.

Several measures are used to reduce the traffic of getattr calls to the server:

• Whenever a new value of Tmserver is received at a client, it is applied to all cache

entries derived from the relevant file.

• The current attribute values are sent ‘piggybacked’ with the results of every

operation on a file, and if the value of Tmserver has changed the client uses it to

update the cache entries relating to the file.

• The adaptive algorithm for setting freshness interval t outlined above reduces the

traffic considerably for most files.

The validation procedure does not guarantee the same level of consistency of files that

is provided in conventional UNIX systems, since recent updates are not always visible

to clients sharing a file. There are two sources of time lag; the delay after a write before

the updated data leaves the cache in the updating client’s kernel and the three-second

‘window’ for cache validation. Fortunately, most UNIX applications do not depend

critically upon the synchronization of file updates, and few difficulties have been

reported from this source.

Writes are handled differently. When a cached page is modified it is marked as

‘dirty’ and is scheduled to be flushed to the server asynchronously. Modified pages are

flushed when the file is closed or a sync occurs at the client, and they are flushed more

frequently if bio-daemons are in use (see below). This does not provide the same

persistence guarantee as the server cache, but it emulates the behaviour for local writes.

To implement read-ahead and delayed-write, the NFS client needs to perform

some reads and writes asynchronously. This is achieved in UNIX implementations of

( ) T Tc – < t ∨ ( ) Tmclient= Tmserver560 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

NFS by the inclusion of one or more bio–daemon processes at each client. (Bio stands

for block input-output; the term daemon is often used to refer to user-level processes that

perform system tasks.) The role of the bio-daemons is to perform read-ahead and

delayed-write operations. A bio-daemon is notified after each read request, and it

requests the transfer of the following file block from the server to the client cache. In the

case of writing, the bio-daemon will send a block to the server whenever a block has

been filled by a client operation. Directory blocks are sent whenever a modification has

occurred.

Bio-daemon processes improve performance, ensuring that the client module does

not block waiting for reads to return or writes to commit at the server. They are not a

logical requirement, since in the absence of read-ahead, a read operation in a user

process will trigger a synchronous request to the relevant server, and the results of writes

in user processes will be transferred to the server when the relevant file is closed or when

the virtual file system at the client performs a sync operation.

Other optimizations • The Sun file system is based on the UNIX BSD Fast File System

which uses 8-kbyte disk blocks, resulting in fewer file system calls for sequential file

access than previous UNIX systems. The UDP packets used for the implementation of

Sun RPC are extended to 9 kilobytes, enabling an RPC call containing an entire block

as an argument to be transferred in a single packet and minimizing the effect of network

latency when reading files sequentially. In NFS version 3, there is no limit on the

maximum size of file blocks that can be handled in read and write operations; clients

and servers can negotiate sizes larger than 8 kbytes if both are able to handle them.

As mentioned above, the file status information cached at clients must be updated

at least every three seconds for active files. To reduce the consequential server load

resulting from getattr requests, all operations that refer to files or directories are taken

as implicit getattr requests, and the current attribute values are ‘piggybacked’ along with

the other results of the operation.

Securing NFS with Kerberos • In Section 11.6.2 we described the Kerberos

authentication system developed at MIT, which has become an industry standard for

securing intranet servers against unauthorized access and imposter attacks. The security

of NFS implementations has been strengthened by the use of the Kerberos scheme to

authenticate clients. In this subsection, we describe the ‘Kerberization’ of NFS as

carried out by the designers of Kerberos.

In the original standard implementation of NFS, the user’s identity is included in

each request in the form of an unencrypted numeric identifier. (The identifier can be

encrypted in later versions of NFS.) NFS does not take any further steps to check the

authenticity of the identifier supplied. This implies a high degree of trust in the integrity

of the client computer and its software by NFS, whereas the aim of Kerberos and other

authentication-based security systems is to reduce to a minimum the range of

components in which trust is assumed. Essentially, when NFS is used in a ‘Kerberized’

environment it should accept requests only from clients whose identity can be shown to

have been authenticated by Kerberos.

One obvious solution considered by the Kerberos developers was to change the

nature of the credentials required by NFS to be a full-blown Kerberos ticket and

authenticator. But because NFS is implemented as a stateless server, each individual file

access request is handled on its face value and the authentication data would have to beSECTION 12.3 CASE STUDY: SUN NETWORK FILE SYSTEM 561

included in each request. This was considered unacceptably expensive in terms of the

time required to perform the necessary encryptions and because it would have entailed

adding the Kerberos client library to the kernel of all workstations.

Instead, a hybrid approach was adopted in which the NFS mount server is supplied

with full Kerberos authentication data for the users when their home and root

filesystems are mounted. The results of this authentication, including the user’s

conventional numerical identifier and the address of the client computer, are retained by

the server with the mount information for each filesystem. (Although the NFS server

does not retain state relating to individual client processes, it does retain the current

mounts at each client computer.)

On each file access request, the NFS server checks the user identifier and the

sender’s address and grants access only if they match those stored at the server for the

relevant client at mount time. This hybrid approach involves only minimal additional

cost and is safe against most forms of attack, provided that only one user at a time can

log in to each client computer. At MIT, the system is configured so that this is the case.

Recent NFS implementations include Kerberos authentication as one of several options

for authentication, and sites that also run Kerberos servers are advised to use this option.

Performance • Early performance figures reported by Sandberg [1987] showed that the

use of NFS did not normally impose a performance penalty in comparison with access

to files stored on local disks. He identified two remaining problem areas:

• frequent use of the getattr call in order to fetch timestamps from servers for cache

validation;

• relatively poor performance of the write operation because write-through was

used at the server.

He noted that writes are relatively infrequent in typical UNIX workloads (about 5% of

all calls to the server), and the cost of write-through is therefore tolerable except when

large files are written to the server. Further, the version of NFS that he tested did not

include the commit mechanism outlined above, which has resulted in a substantial

improvement in write performance in current versions. His results also show that the

lookup operation accounts for almost 50% of server calls. This is a consequence of the

step-by-step pathname translation method necessitated by UNIX’s file-naming

semantics.

Measurements are taken regularly by Sun and other NFS implementors using an

updated version of an exhaustive set of benchmark programs known as LADDIS [Keith

and Wittle 1993]. Current and past results are available at the SPEC web site

[www.spec.org]. Performance is summarized there for NFS server implementations

from many vendors and different hardware configurations. Single-CPU

implementations based on PC hardware but with dedicated operating systems achieve

throughputs in excess of 12,000 server operations per second and large multi-processor

configurations with many disks and controllers have achieved throughputs of up to

300,000 server operations per second. These figures indicate that NFS offers a very

effective solution to distributed storage needs in intranets of most sizes and types of use,

ranging for example from a traditional UNIX load of development by several hundred

software engineers to a battery of web servers serving material from an NFS server.562 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

NFS summary • Sun NFS closely follows our abstract model. The resulting design

provides good location and access transparency if the NFS mount service is used

properly to produce similar name spaces at all clients. NFS supports heterogeneous

hardware and operating systems. The NFS server implementation is stateless, enabling

clients and servers to resume execution after a failure without the need for any recovery

procedures. Migration of files or filesystems is not supported, except at the level of

manual intervention to reconfigure mount directives after the movement of a filesystem

to a new location.

The performance of NFS is much enhanced by the caching of file blocks at each

client computer. This is important for the achievement of satisfactory performance but

results in some deviation from strict UNIX one-copy file update semantics.

The other design goals of NFS and the extent to which they have been achieved are

discussed below.

Access transparency: The NFS client module provides an application programming

interface to local processes that is identical to the local operating system’s interface.

Thus in a UNIX client, accesses to remote files are performed using the normal UNIX

system calls. No modifications to existing programs are required to enable them to

operate correctly with remote files.

Location transparency: Each client establishes a file name space by adding mounted

directories in remote filesystems to its local name space. File systems have to be

exported by the node that holds them and remote-mounted by a client before they can

be accessed by processes running in the client (see Figure 12.10). The point in a

client’s name hierarchy at which a remote-mounted file system appears is determined

by the client; NFS does not enforce a single network-wide file name space – each

client sees a set of remote filesystems that is determined locally, and remote files may

have different pathnames on different clients, but a uniform name space can be

established with appropriate configuration tables in each client, achieving the goal of

location transparency.

Mobility transparency: Filesystems (in the UNIX sense, that is, subtrees of files)

may be moved between servers, but the remote mount tables in each client must then

be updated separately to enable the clients to access the filesystems in their new

locations, thus migration transparency is not fully achieved by NFS.

Scalability: The published performance figures show that NFS servers can be built

to handle very large real-world loads in an efficient and cost-effective manner. The

performance of a single server can be increased by the addition of processors, disks

and controllers. When the limits of that process are reached, additional servers must

be installed and the filesystems must be reallocated between them. The effectiveness

of that strategy is limited by the existence of ‘hot spot’ files – single files that are

accessed so frequently that the server reaches a performance limit. When loads

exceed the maximum performance available with that strategy, a distributed file

system that supports replication of updatable files (such as Coda, described in

Chapter 18), or one such as AFS that reduces the protocol traffic by the caching of

whole files, may offer a better solution. We discuss other approaches to scalability in

Section 12.5.SECTION 12.3 CASE STUDY: SUN NETWORK FILE SYSTEM 563

File replication: Read-only file stores can be replicated on several NFS servers, but

NFS does not support file replication with updates. The Sun Network Information

Service (NIS) is a separate service available for use with NFS that supports the

replication of simple databases organized as key-value pairs (for example, the UNIX

system files /etc/passwd and /etc/hosts). It manages the distribution of updates and

accesses to the replicated files based on a simple master–slave replication model

(also known as the primary copy model, discussed further in Chapter 18) with

provision for the replication of part or all of the database at each site. NIS provides a

shared repository for system information that changes infrequently and does not

require updates to occur simultaneously at all sites.

Hardware and operating system heterogeneity: NFS has been implemented for

almost every known operating system and hardware platform and is supported by a

variety of filing systems.

Fault tolerance: The stateless and idempotent nature of the NFS file access protocol

ensures that the failure modes observed by clients when accessing remote files are

similar to those for local file access. When a server fails, the service that it provides

is suspended until the server is restarted, but once it has been restarted user-level

client processes proceed from the point at which the service was interrupted, unaware

of the failure (except in the case of access to soft-mounted remote file systems). In

practice, hard mounting is used in most instances, and this tends to impede

application programs handling server failures gracefully.

The failure of a client computer or a user-level process in a client has no effect

on any server that it may be using, since servers hold no state on behalf of their

clients.

Consistency: We have described the update behaviour in some detail. It provides a

close approximation to one-copy semantics and meets the needs of the vast majority

of applications, but the use of file sharing via NFS for communication or close

coordination between processes on different computers cannot be recommended.

Security: The need for security in NFS emerged with the connection of most

intranets to the Internet. The integration of Kerberos with NFS was a major step

forward. Other recent developments include the option to use a secure RPC

implementation (RPCSEC\_GSS, documented in RFC 2203 [Eisler et al. 1997]) for

authentication and to ensure the privacy and security of the data transmitted with read

and write operations. Installations that have not deployed these mechanisms abound,

though, and they are insecure.

Efficiency: The measured performance of several implementations of NFS and its

widespread adoption for use in situations that generate very heavy loads are clear

indications of the efficiency with which the NFS protocol can be implemented.564 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

12.4 Case study: The Andrew File System

Like NFS, AFS provides transparent access to remote shared files for UNIX programs

running on workstations. Access to AFS files is via the normal UNIX file primitives,

enabling existing UNIX programs to access AFS files without modification or

recompilation. AFS is compatible with NFS. AFS servers hold ‘local’ UNIX files, but

the filing system in the servers is NFS-based, so files are referenced by NFS-style file

handles rather than i-node numbers, and the files may be remotely accessed via NFS.

AFS differs markedly from NFS in its design and implementation. The differences

are primarily attributable to the identification of scalability as the most important design

goal. AFS is designed to perform well with larger numbers of active users than other

distributed file systems. The key strategy for achieving scalability is the caching of

whole files in client nodes. AFS has two unusual design characteristics:

Whole-file serving: The entire contents of directories and files are transmitted to

client computers by AFS servers (in AFS-3, files larger than 64 kbytes are

transferred in 64-kbyte chunks).

Whole-file caching: Once a copy of a file or a chunk has been transferred to a

client computer it is stored in a cache on the local disk. The cache contains several

hundred of the files most recently used on that computer. The cache is permanent,

surviving reboots of the client computer. Local copies of files are used to satisfy

clients’ open requests in preference to remote copies whenever possible.

Scenario • Here is a simple scenario illustrating the operation of AFS:

1. When a user process in a client computer issues an open system call for a file in

the shared file space and there is not a current copy of the file in the local cache,

the server holding the file is located and is sent a request for a copy of the file.

2. The copy is stored in the local UNIX file system in the client computer. The copy

is then opened and the resulting UNIX file descriptor is returned to the client.

3. Subsequent read, write and other operations on the file by processes in the client

computer are applied to the local copy.

4. When the process in the client issues a close system call, if the local copy has been

updated its contents are sent back to the server. The server updates the file

contents and the timestamps on the file. The copy on the client’s local disk is

retained in case it is needed again by a user-level process on the same workstation.

We discuss the observed performance of AFS below, but we can make some general

observations and predictions here based on the design characteristics described above:

• For shared files that are infrequently updated (such as those containing the code

of UNIX commands and libraries) and for files that are normally accessed by only

a single user (such as most of the files in a user’s home directory and its subtree),

locally cached copies are likely to remain valid for long periods – in the first case

because they are not updated and in the second because if they are updated, the

updated copy will be in the cache on the owner’s workstation. These classes of file

account for the overwhelming majority of file accesses.SECTION 12.4 CASE STUDY: THE ANDREW FILE SYSTEM 565

• The local cache can be allocated a substantial proportion of the disk space on each

workstation – say, 100 megabytes. This is normally sufficient for the establishment

of a working set of the files used by one user. The provision of sufficient cache

storage for the establishment of a working set ensures that files in regular use on a

given workstation are normally retained in the cache until they are needed again.

• The design strategy is based on some assumptions about average and maximum

file size and locality of reference to files in UNIX systems. These assumptions are

derived from observations of typical UNIX workloads in academic and other

environments [Satyanarayanan 1981, Ousterhout et al. 1985, Floyd 1986]. The

most important observations are:

– Files are small; most are less than 10 kilobytes in size.

– Read operations on files are much more common than writes (about six times

more common).

– Sequential access is common, and random access is rare.

– Most files are read and written by only one user. When a file is shared, it is

usually only one user who modifies it.

– Files are referenced in bursts. If a file has been referenced recently, there is a

high probability that it will be referenced again in the near future.

These observations were used to guide the design and optimization of AFS, not to

restrict the functionality seen by users.

• AFS works best with the classes of file identified in the first point above. There is

one important type of file that does not fit into any of these classes – databases are

typically shared by many users and are often updated quite frequently. The

designers of AFS have explicitly excluded the provision of storage facilities for

databases from their design goals, stating that the constraints imposed by different

naming structures (that is, content-based access) and the need for fine-grained data

access, concurrency control and atomicity of updates make it difficult to design a

distributed database system that is also a distributed file system. They argue that

the provision of facilities for distributed databases should be addressed separately

[Satyanarayanan 1989a].

12.4.1 Implementation

The above scenario illustrates AFS’s operation but leaves many questions about its

implementation unanswered. Among the most important are:

• How does AFS gain control when an open or close system call referring to a file

in the shared file space is issued by a client?

• How is the server holding the required file located?

• What space is allocated for cached files in workstations?

• How does AFS ensure that the cached copies of files are up-to-date when files may

be updated by several clients?

We answer these questions below.566 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

AFS is implemented as two software components that exist as UNIX processes

called Vice and Venus. Figure 12.11 shows the distribution of Vice and Venus processes.

Vice is the name given to the server software that runs as a user-level UNIX process in

each server computer, and Venus is a user-level process that runs in each client computer

and corresponds to the client module in our abstract model.

The files available to user processes running on workstations are either local or

shared. Local files are handled as normal UNIX files. They are stored on a workstation’s

disk and are available only to local user processes. Shared files are stored on servers, and

copies of them are cached on the local disks of workstations. The name space seen by

user processes is illustrated in Figure 12.12. It is a conventional UNIX directory

hierarchy, with a specific subtree (called cmu) containing all of the shared files. This

splitting of the file name space into local and shared files leads to some loss of location

transparency, but this is hardly noticeable to users other than system administrators.

Local files are used only for temporary files (/tmp) and processes that are essential for

workstation startup. Other standard UNIX files (such as those normally found in /bin,

/lib and so on) are implemented as symbolic links from local directories to files held in

the shared space. Users’ directories are in the shared space, enabling users to access their

files from any workstation.

The UNIX kernel in each workstation and server is a modified version of BSD

UNIX. The modifications are designed to intercept open, close and some other file

system calls when they refer to files in the shared name space and pass them to the Venus

process in the client computer (illustrated in Figure 12.13). One other kernel

modification is included for performance reasons, and this is described later.

One of the file partitions on the local disk of each workstation is used as a cache,

holding the cached copies of files from the shared space. Venus manages the cache,

removing the least recently used files when a new file is acquired from a server to make

Figure 12.11 Distribution of processes in the Andrew File System

Venus

Workstations Servers

Venus

User Venus

program

Network

UNIX kernel

UNIX kernel

Vice

User

program

User

program

Vice

UNIX kernel

UNIX kernel

UNIX kernelSECTION 12.4 CASE STUDY: THE ANDREW FILE SYSTEM 567

the required space if the partition is full. The workstation cache is usually large enough

to accommodate several hundred average-sized files, rendering the workstation largely

independent of the Vice servers once a working set of the current user’s files and

frequently used system files has been cached.

AFS resembles the abstract file service model described in Section 12.2 in these

respects:

• A flat file service is implemented by the Vice servers, and the hierarchic directory

structure required by UNIX user programs is implemented by the set of Venus

processes in the workstations.

• Each file and directory in the shared file space is identified by a unique, 96-bit file

identifier (fid) similar to a UFID. The Venus processes translate the pathnames

issued by clients to fids.

Figure 12.12 File name space seen by clients of AFS

/ (root)

tmp bin . . . vmunix cmu

bin

Local Shared

Symbolic

links

Figure 12.13 System call interception in AFS

UNIX file

system calls

Non-local file

operations

Workstation

Local

disk

User

program

UNIX kernel

Venus

UNIX file system

Venus568 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

Files are grouped into volumes for ease of location and movement. Volumes are

generally smaller than the UNIX filesystems, which are the unit of file grouping in NFS.

For example, each user’s personal files are generally located in a separate volume. Other

volumes are allocated for system binaries, documentation and library code.

The representation of fids includes the volume number for the volume containing

the file (cf. the file group identifier in UFIDs), an NFS file handle identifying the file

within the volume (cf. the file number in UFIDs) and a uniquifier to ensure that file

identifiers are not reused:

User programs use conventional UNIX pathnames to refer to files, but AFS uses fids in

the communication between the Venus and Vice processes. The Vice servers accept

requests only in terms of fids. Venus translates the pathnames supplied by clients into

fids using a step-by-step lookup to obtain the information from the file directories held

in the Vice servers.

Figure 12.14 describes the actions taken by Vice, Venus and the UNIX kernel

when a user process issues each of the system calls mentioned in our outline scenario

above. The callback promise mentioned here is a mechanism for ensuring that cached

copies of files are updated when another client closes the same file after updating it. This

mechanism is discussed in the next section.

12.4.2 Cache consistency

When Vice supplies a copy of a file to a Venus process it also provides a callback

promise – a token issued by the Vice server that is the custodian of the file, guaranteeing

that it will notify the Venus process when any other client modifies the file. Callback

promises are stored with the cached files on the workstation disks and have two states:

valid or cancelled. When a server performs a request to update a file it notifies all of the

Venus processes to which it has issued callback promises by sending a callback to each

– a callback is a remote procedure call from a server to a Venus process. When the Venus

process receives a callback, it sets the callback promise token for the relevant file to

cancelled.

Whenever Venus handles an open on behalf of a client, it checks the cache. If the

required file is found in the cache, then its token is checked. If its value is cancelled, then

a fresh copy of the file must be fetched from the Vice server, but if the token is valid,

then the cached copy can be opened and used without reference to Vice.

When a workstation is restarted after a failure or a shutdown, Venus aims to retain

as many as possible of the cached files on the local disk, but it cannot assume that the

callback promise tokens are correct, since some callbacks may have been missed. Before

the first use of each cached file or directory after a restart, Venus therefore generates a

cache validation request containing the file modification timestamp to the server that is

the custodian of the file. If the timestamp is current, the server responds with valid and

the token is reinstated. If the timestamp shows that the file is out of date, then the server

responds with cancelled and the token is set to cancelled. Callbacks must be renewed

32 bits 32 bits 32 bits

Volume number File handle UniquifierSECTION 12.4 CASE STUDY: THE ANDREW FILE SYSTEM 569

before an open if a time T (typically on the order of a few minutes) has elapsed since the

file was cached without communication from the server. This is to deal with possible

communication failures, which can result in the loss of callback messages.

This callback-based mechanism for maintaining cache consistency was adopted as

offering the most scalable approach, following the evaluation in the prototype (AFS-1)

of a timestamp-based mechanism similar to that used in NFS. In AFS-1, a Venus process

holding a cached copy of a file interrogates the Vice process on each open to determine

whether the timestamp on the local copy agrees with that on the server. The callbackbased approach is more scalable because it results in communication between client and

server and activity in the server only when the file has been updated, whereas the

timestamp approach results in a client-server interaction on each open, even when there

is a valid local copy. Since the majority of files are not accessed concurrently, and read

operations predominate over writes in most applications, the callback mechanism

results in a dramatic reduction in the number of client-server interactions.

The callback mechanism used in AFS-2 and later versions of AFS requires Vice

servers to maintain some state on behalf of their Venus clients, unlike AFS-1, NFS and

our file service model. The client-dependent state required consists of a list of the Venus

Figure 12.14 Implementation of file system calls in AFS

User process UNIX kernel Venus Net Vice

open(FileName,

mode)

If FileName refers to a

file in shared file space,

pass the request to

Venus.

Open the local file and

return the file

descriptor to the

application.

Check list of files in

local cache. If not

present or there is no

valid callback promise,

send a request for the

file to the Vice server

that is custodian of the

volume containing the

file.

Place the copy of the

file in the local file

system, enter its local

name in the local cache

list and return the local

name to UNIX.

Transfer a copy of the

file and a callback

promise to the

workstation. Log the

callback promise.

read(FileDescriptor,

Buffer, length)

Perform a normal

UNIX read operation

on the local copy.

write(FileDescriptor,

Buffer, length)

Perform a normal

UNIX write operation

on the local copy.

close(FileDescriptor) Close the local copy

and notify Venus that

the file has been closed. If the local copy has

been changed, send a

copy to the Vice server

that is the custodian of

the file.

Replace the file

contents and send a

callback to all other

clients holding callback

promises on the file.570 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

processes to which callback promises have been issued for each file. These callback lists

must be retained over server failures – they are held on the server disks and are updated

using atomic operations.

Figure 12.15 shows the RPC calls provided by AFS servers for operations on files

(that is, the interface provided by AFS servers to Venus processes).

Update semantics • The goal of this cache-consistency mechanism is to achieve the

best approximation to one-copy file semantics that is practicable without serious

performance degradation. A strict implementation of one-copy semantics for UNIX file

access primitives would require that the results of each write to a file be distributed to

all sites holding the file in their cache before any further accesses can occur. This is not

practicable in large-scale systems; instead, the callback promise mechanism maintains

a well-defined approximation to one-copy semantics.

For AFS-1, the update semantics can be formally stated in very simple terms. For

a client C operating on a file F whose custodian is a server S, the following guarantees

of currency for the copies of F are maintained:

after a successful open: latest(F, S)

after a failed open: failure(S)

after a successful close: updated(F, S)

after a failed close: failure(S)

where latest(F, S) denotes a guarantee that the current value of F at C is the same as the

value at S, failure(S) denotes that the open or close operation has not been performed at

Figure 12.15 The main components of the Vice service interface

Note: Directory and administrative operations (Rename, Link, Makedir, Removedir, GetTime,

CheckToken and so on) are not shown.

Fetch(fid) → attr, data Returns the attributes (status) and, optionally, the contents of

the file identified by fid and records a callback promise on it.

Store(fid, attr, data) Updates the attributes and (optionally) the contents of a

specified file.

Create() → fid Creates a new file and records a callback promise on it.

Remove(fid) Deletes the specified file.

SetLock(fid, mode) Sets a lock on the specified file or directory. The mode of the

lock may be shared or exclusive. Locks that are not removed

expire after 30 minutes.

ReleaseLock(fid) Unlocks the specified file or directory.

RemoveCallback(fid) Informs the server that a Venus process has flushed a file from

its cache.

BreakCallback(fid) Call made by a Vice server to a Venus process; cancels the

callback promise on the relevant file.SECTION 12.4 CASE STUDY: THE ANDREW FILE SYSTEM 571

S (and the failure can be detected by C), and updated(F, S) denotes that C’s value of F

has been successfully propagated to S.

For AFS-2, the currency guarantee for open is slightly weaker, and the

corresponding formal statement of the guarantee is more complex. This is because a

client may open an old copy of a file after it has been updated by another client. This

occurs if a callback message is lost, for example as a result of a network failure. But

there is a maximum time, T, for which a client can remain unaware of a newer version

of a file. Hence we have the following guarantee:

after a successful open: latest(F, S, 0)

or (lostCallback(S, T) and inCache(F) and

latest(F, S, T))

where latest(F, S, T) denotes that the copy of F seen by the client is no more than T

seconds out of date, lostCallback(S, T) denotes that a callback message from S to C has

been lost at some time during the last T seconds, and inCache(F) indicates that the file

F was in the cache at C before the open operation was attempted. The above formal

statement expresses the fact that the cached copy of F at C after an open operation is the

most recent version in the system or a callback message has been lost (due to a

communication failure) and the version that was already in the cache has been used; the

cached version will be no more than T seconds out of date. (T is a system constant

representing the interval at which callback promises must be renewed. At most

installations, the value of T is about 10 minutes.)

In line with its goal – to provide a large-scale, UNIX-compatible distributed file

service – AFS does not provide any further mechanism for the control of concurrent

updates. The cache consistency algorithm described above comes into action only on

open and close operations. Once a file has been opened, the client may access and update

the local copy in any way it chooses without the knowledge of any processes on other

workstations. When the file is closed, a copy is returned to the server, replacing the

current version.

If clients in different workstations open, write and close the same file

concurrently, all but the update resulting from the last close will be silently lost (no error

report is given). Clients must implement concurrency control independently if they

require it. On the other hand, when two client processes in the same workstation open a

file, they share the same cached copy and updates are performed in the normal UNIX

fashion – block by block.

Although the update semantics differ depending on the locations of the concurrent

processes accessing a file and are not precisely the same as those provided by the

standard UNIX file system, they are sufficiently close for the vast majority of existing

UNIX programs to operate correctly.

12.4.3 Other aspects

AFS introduces several other interesting design developments and refinements that we

outline here, together with a summary of performance evaluation results:

UNIX kernel modifications • We have noted that the Vice server is a user-level process

running in the server computer and the server host is dedicated to the provision of an572 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

AFS service. The UNIX kernel in AFS hosts is altered so that Vice can perform file

operations in terms of file handles instead of the conventional UNIX file descriptors.

This is the only kernel modification required by AFS, and it is necessary if Vice is not

to maintain any client state (such as file descriptors).

Location database • Each server contains a copy of a fully replicated location database

giving a mapping of volume names to servers. Temporary inaccuracies in this database

may occur when a volume is moved, but they are harmless because forwarding

information is left behind in the server from which the volume is moved.

Threads • The implementations of Vice and Venus make use of a non-preemptive

threads package to enable requests to be processed concurrently at both the client (where

several user processes may have file access requests in progress concurrently) and the

server. In the client, the tables describing the contents of the cache and the volume

database are held in memory that is shared between the Venus threads.

Read-only replicas • Volumes containing files that are frequently read but rarely

modified, such as the UNIX /bin and /usr/bin directories of system commands and /man

directory of manual pages, can be replicated as read-only volumes at several servers.

When this is done, there is only one read-write replica and all updates are directed to it.

The propagation of the changes to the read-only replicas is performed after the update

by an explicit operational procedure. Entries in the location database for volumes that

are replicated in this way are one-to-many, and the server for each client request is

selected on the bases of server loads and accessibility.

Bulk transfers • AFS transfers files between clients and servers in 64-kilobyte chunks.

The use of such a large packet size is an important aid to performance, minimizing the

effect of network latency. Thus the design of AFS enables the use of the network to be

optimized.

Partial file caching • The need to transfer the entire contents of files to clients even

when the application requirement is to read only a small portion of the file is an obvious

source of inefficiency. Version 3 of AFS removed this requirement, allowing file data

to be transferred and cached in 64-kbyte blocks while still retaining the consistency

semantics and other features of the AFS protocol.

Performance • The primary goal of AFS is scalability, so its performance with large

numbers of users is of particular interest. Howard et al. [1988] give details of extensive

comparative performance measurements, which were undertaken using a specially

developed AFS benchmark that has subsequently been widely used for the evaluation of

distributed file systems. Not surprisingly, whole-file caching and the callback protocol

led to dramatically reduced loads on the servers. Satyanarayanan [1989a] states that a

server load of 40% was measured with 18 client nodes running a standard benchmark,

against a load of 100% for NFS running the same benchmark. Satyanarayanan attributes

much of the performance advantage of AFS to the reduction in server load deriving from

the use of callbacks to notify clients of updates to files, compared with the timeout

mechanism used in NFS for checking the validity of pages cached at clients.

Wide area support: • Version 3 of AFS supports multiple administrative cells, each

with its own servers, clients, system administrators and users. Each cell is a completely

autonomous environment, but a federation of cells can cooperate in presenting usersSECTION 12.5 ENHANCEMENTS AND FURTHER DEVELOPMENTS 573

with a uniform, seamless file name space. The resulting system was widely deployed by

the Transarc Corporation, and a detailed survey of the resulting performance usage

patterns was published [Spasojevic and Satyanarayanan 1996]. The system was installed

on over 1000 servers at over 150 sites. The survey showed cache hit ratios in the range

of 96 –98% for accesses to a sample of 32,000 file volumes holding 200 Gbytes of data.

12.5 Enhancements and further developments

Several advances have been made in the design of distributed file systems since the

emergence of NFS and AFS. In this section, we describe advances that enhance the

performance, availability and scalability of conventional distributed file systems. More

radical advances are described elsewhere in the book, including the maintenance of

consistency in replicated read-write filesystems to support disconnected operation and

high availability in the Bayou and Coda systems (Sections 18.4.2 and 18.4.3) and a

highly scalable architecture for the delivery of streams of real-time data with quality

guarantees in the Tiger video file server (Section 20.6.1).

NFS enhancements • Several research projects have addressed the need for one-copy

update semantics by extending the NFS protocol to include open and close operations

and adding a callback mechanism to enable the server to notify clients of the need to

invalidate cache entries. We describe two such efforts here; their results seem to indicate

that these enhancements can be accommodated without undue complexity or extra

communication costs.

Some recent efforts by Sun and other NFS developers have been directed at

making NFS servers more accessible and useful in wide-area networks. While the HTTP

protocol supported by web servers offers an effective and highly scalable method for

making whole files available to clients throughout the Internet, it is less useful to

application programs that require access to portions of large files or those that update

portions of files. The WebNFS development (described below) makes it possible for

application programs to become clients of NFS servers anywhere in the Internet (using

the NFS protocol directly instead of indirectly through a kernel module). This, together

with appropriate libraries for Java and other network programming languages, should

offer the possibility of implementing Internet applications that share data directly, such

as multi-user games or clients of large dynamic databases.

Achieving one-copy update semantics: The stateless server architecture of NFS brought

great advantages in terms of robustness and ease of implementation, but it precluded the

achievement of precise one-copy update semantics (the effects of concurrent writes by

different clients to the same file are not guaranteed to be the same as they would be in a

single UNIX system when multiple processes write to a local file). It also prevents the

use of callbacks notifying clients of changes to files, and this results in frequent getattr

requests from clients to check for file modification.

Two research systems have been developed that address these drawbacks. Spritely

NFS [Srinivasan and Mogul 1989, Mogul 1994] is a version of the file system developed

for the Sprite distributed operating system at Berkeley [Nelson et al. 1988]. Spritely

NFS is an implementation of the NFS protocol with the addition of open and close calls.574 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

Clients’ modules must send an open operation whenever a local user-level process opens

a file that is on the server. The parameters of the Sprite open operation specify a mode

(read, write or both) and include counts of the number of local processes that currently

have the file open for reading and for writing. Similarly, when a local process closes a

remote file, a close operation is sent to the server with updated counts of readers and

writers. The server records these numbers in an open files table with the IP address and

port number of the client.

When the server receives an open, it checks the open files table for other clients

that have the same file open and sends callback messages to those clients instructing

them to modify their caching strategy. If the open specifies write mode, then it will fail

if any other client has the file open for writing. Other clients that have the file open for

reading will be instructed to invalidate any locally cached portions of the file.

For open operations that specify read mode, the server sends a callback message

to any client that is writing, instructing it to stop caching (i.e., to use a strictly writethrough mode of operation), and it instructs all clients that are reading to cease caching

the file (so that all local read calls result in a request to the server).

These measures result in a file service that maintains the UNIX one-copy update

semantics at the expense of carrying some client-related state at the server. They also

enable some efficiency gains in the handling of cached writes. If the client-related state

is held in volatile memory at the server, it is vulnerable to server crashes. Spritely NFS

implements a recovery protocol that interrogates a list of clients that have recently

opened files on the server to recover the full open files table. The list of clients is stored

on disk, is updated relatively infrequently and is ‘pessimistic’ – it may safely include

more clients than those that had files open at the time of a crash. Failed clients may also

result in excess entries in the open files table, but these entries will be removed when the

clients restart.

When Spritely NFS was evaluated against NFS version 2, it showed a modest

performance improvement. This was due to the improved caching of writes. Changes in

NFS version 3 would probably result in at least as great an improvement, but the results

of the Spritely NFS project certainly indicate that it is possible to achieve one-copy

update semantics without substantial loss of performance, albeit at the expense of some

extra implementation complexity in the client and server modules and the need for a

recovery mechanism to restore the state after a server crash.

NQNFS: The NQNFS (Not Quite NFS) project [Macklem 1994] had similar aims to

Spritely NFS – to add more precise cache consistency to the NFS protocol and to

improve performance through better use of caching. An NQNFS server maintains

similar client-related state concerning open files, but it uses leases (Section 5.4.3) to aid

recovery after a server crash. The server sets an upper bound on the time for which a

client may hold a lease on an open file. If the client wishes to continue beyond that time,

it must renew the lease. Callbacks are used in a similar manner to Spritely NFS to

request clients to flush their caches when a write request occurs, but if the clients don’t

reply, the server simply waits until their leases expire before responding to the new write

request.

WebNFS: The advent of the Web and Java applets led to the recognition by the NFS

development team and others that some Internet applications could benefit from directSECTION 12.5 ENHANCEMENTS AND FURTHER DEVELOPMENTS 575

access to NFS servers without many of the overheads associated with the emulation of

UNIX file operations included in standard NFS clients.

The aim of WebNFS (described in RFCs 2055 and 2056 [Callaghan 1996a,

1996b]) is to enable web browsers and other applications to access files on an NFS

server that ‘publishes’ them using a public file handle relative to a public root directory.

This mode of use bypasses the mount service and the port mapper service (described in

Chapter 5). WebNFS clients interact with an NFS server at a well-known port number

(2049). To access files by pathname, they issue lookup requests using a public file

handle. The public file handle has a well-known value that is interpreted specially by the

virtual file system at the server. Because of the high latency of wide-area networks, a

multicomponent variant of the lookup operation is used to look up a multi-part pathname

in a single request.

Thus WebNFS enables clients to be written that access portions of files stored in

NFS servers at remote sites with minimal setup overheads. There is provision for access

control and authentication, but in many cases the client will require only read access to

public files, and in that case the authentication option can be turned off. To read a

portion of a single file located on an NFS server that supports WebNFS requires the

establishment of a TCP connection and two RPC calls – a multicomponent lookup and

a read operation. The size of the block of data read is not limited by the NFS protocol.

For example, a weather service might publish a file on its NFS server containing

a large database of frequently updated weather data with a URL such as:

nfs://data.weather.gov/weatherdata/global.data

An interactive WeatherMap client, that displays weather maps could be constructed in

Java or any other language that supports a WebNFS procedure library. The client reads

only those portions of the /weatherdata/global.data file that are needed to construct the

particular maps requested by a user, whereas a similar application that used HTTP to

access a weather data server either would have to transfer the entire database to the client

or would require the support of a special-purpose server program to supply it with the

data it requires.

NFS version 4: A new version of the NFS protocol was introduced in 2000. The goals of

NFS version 4 are described in RFC 2624 [Shepler 1999] and in Brent Callaghan’s book

[Callaghan 1999]. Like WebNFS, it aims to make it practical to use NFS in wide-area

networks and Internet applications. It includes the features of WebNFS, but the

introduction of a new protocol also offers an opportunity to make more radical

enhancements. (WebNFS was restricted to changes to the server that did not involve the

addition of new operations to the protocol.)

NFS version 4 exploits results that have emerged from research in file server

design over the past decade, such as the use of callbacks or leases to maintain

consistency. NFS version 4 supports on-the-fly recovery from server faults by allowing

file systems to be moved to new servers transparently. Scalability is improved by using

proxy servers in a manner analogous to their use in the Web.

AFS enhancements • We have mentioned that DCE/DFS, the distributed file system

included in the Open Software Foundation’s Distributed Computing Environment

[www.opengroup.org], was based on the Andrew File System. The design of DCE/DFS

goes beyond AFS, particularly in its approach to cache consistency. In AFS, callbacks576 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

are generated only when the server receives a close operation for a file that has been

updated. DFS adopted a similar strategy to Spritely NFS and NQNFS to generating

callbacks as soon as a file is updated. In order to update a file, a client must obtain a write

token from the server, specifying a range of bytes in the file that the client is permitted

to update. When a write token is requested, clients holding copies of the same file for

reading receive revocation callbacks. Tokens of other types are used to achieve

consistency for cached file attributes and other metadata. All tokens have an associated

lifetime, and clients must renew them after their lifetime has expired.

Improvements in storage organization • There has been considerable progress in the

organization of file data stored on disks. The impetus for much of this work arose from

the increased loads and greater reliability that distributed file systems need to support,

and they have resulted in file systems with substantially improved performance. The

principal results of this work are:

Redundant Arrays of Inexpensive Disks (RAID): This is a mode of storage

[Patterson et al. 1988, Chen et al. 1994] in which data blocks are segmented into

fixed-size chunks and stored in ‘stripes’ across several disks, along with redundant

error-correcting codes that enable the data blocks to be reconstructed completely and

operation to continue normally in the event of disk failures. RAID also produces

considerably better performance than a single disk, because the stripes that make up

a block are read and written concurrently.

Log-structured file storage (LFS): Like Spritely NFS, this technique originated in

the Sprite distributed operating system project at Berkeley [Rosenblum and

Ousterhout 1992]. The authors observed that as larger amounts of main memory

became available for caching in file servers, an increased level of cache hits resulted

in excellent read performance, but write performance remained mediocre. This arose

from the high latencies associated with writing individual data blocks to disk and

associated updates to metadata blocks (that is, the blocks known as i-nodes that hold

file attributes and a vector of pointers to the blocks in a file).

The LFS solution is to accumulate a set of writes in memory and then commit

them to disk in large, contiguous, fixed-sized segments. These are called log

segments because the data and metadata blocks are stored strictly in the order in

which they were updated. A log segment is 1 Mbyte or larger in size and is stored in

a single disk track, removing the disk head latencies associated with writing

individual blocks. Fresh copies of updated data and metadata blocks are always

written, requiring the maintenance of a dynamic map (in memory with a persistent

backup) pointing to the i-node blocks. Garbage collection of stale blocks is also

required, with compaction of ‘live’ blocks to leave contiguous areas of storage free

for the storage of log segments. The latter is a fairly complex process; it is carried out

as a background activity by a component called the cleaner. Some sophisticated

cleaner algorithms have been developed for it based on the results of simulations.

Despite these extra costs, the overall performance gain is outstanding;

Rosenblum and Ousterhout measured a write throughput as high as 70% of the

available disk bandwidth, compared with less than 10% for a conventional UNIX file

system. The log structure also simplifies recovery after server crashes. The Zebra file

system [Hartman and Ousterhout 1995], developed as a follow-on to the original LFSSECTION 12.5 ENHANCEMENTS AND FURTHER DEVELOPMENTS 577

work, combines log-structured writes with a distributed RAID approach – the log

segments are subdivided into sections with error-correcting data and written to disks

on separate network nodes. Performance four to five times better than that of NFS is

claimed for writing large files, with smaller gains for short files.

New design approaches • The availability of high-performance switched networks

(such as ATM and switched high-speed Ethernet) have prompted several efforts to

provide persistent storage systems that distribute file data in a highly scalable and faulttolerant manner among many nodes on an intranet, separating the responsibilities for

reading and writing data from the responsibilities for managing the metadata and

servicing client requests. In the following, we outline two such developments.

These approaches scale better than the more centralized servers that we have

described in the preceding sections. They generally demand a high level of trust among

the computers that cooperate to provide the service, because they include a fairly lowlevel protocol for communication with the nodes holding data (somewhat analogous to

a ‘virtual disk’ API). Hence their scope is likely to be limited to a single local network.

xFS: A group at the University of California, Berkeley, proposed a serverless network

file system architecture and developed a prototype implementation called xFS

[Anderson et al. 1996]. Their approach was motivated by three factors:

1. the opportunity provided by fast switched LANs for multiple file servers in a local

network to transfer bulk data to clients concurrently;

2. increased demand for access to shared data;

3. the fundamental limitations of systems based on central file servers.

Concerning (3), they refer to the facts that the construction of high-performance NFS

servers requires relatively costly hardware with multiple CPUs, disks and network

controllers, and that there are limits to the process of partitioning the file space – i.e.,

placing shared files in separate filesystems mounted on different servers. They also

point to the fact that a central server represents a single point of failure.

xFS is ‘serverless’ in the sense that it distributes file server processing

responsibilities across a set of available computers in a local network at the granularity

of individual files. Storage responsibilities are distributed independently of management

and other service responsibilities: xFS implements a software RAID storage system,

striping file data across disks on multiple computers (in this regard it is a precursor to

the Tiger video file server described in Chapter 20), together with a log-structuring

technique similar to the Zebra file system.

Responsibility for the management of each file can be allocated to any of the

computers supporting the xFS service. This is achieved through a metadata structure

called the manager map, which is replicated at all clients and servers. File identifiers

include a field that acts as an index into the manager map, and each entry in the map

identifies the computer that is currently responsible for managing the corresponding file.

Several other metadata structures, similar to those found in other log-structured and

RAID storage systems, are used for the management of the log-structured file storage

and the striped disk storage.

Anderson et al. constructed a preliminary prototype of xFS and evaluated its

performance. The prototype was incomplete at the time the evaluation was carried out –578 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

the implementation of crash recovery was unfinished and the log-structured storage

scheme lacked a cleaner component to recover space occupied by stale logs and compact

files.

The performance evaluations carried out with this preliminary prototype used 32

single-processor and dual-processor Sun SPARCstations connected to a high-speed

network. The evaluations compared the xFS file service running on up to 32

workstations with NFS and with AFS, each running on a single dual-processor Sun

SPARCStation. The read and write bandwidths achieved with xFS with 32 servers

exceeded those of NFS and AFS with a single dual-processor server by approximately

a factor of 10. The difference in performance was much less marked when xFS was

compared with NFS and AFS using the standard AFS benchmark. But overall, the

results indicate that the highly distributed processing and storage architecture of xFS

offers a promising direction for achieving better scalability in distributed file systems.

Frangipani: Frangipani is a highly scalable distributed file system developed and

deployed at the Digital Systems Research Center (now Compaq Systems Research

Center) [Thekkath et al. 1997]. Its goals are very similar to those of xFS, and like xFS,

it approaches them with a design that separates persistent storage responsibilities from

other file service actions. But Frangipani’s service is structured as two totally

independent layers. The lower layer is provided by the Petal distributed virtual disk

system [Lee and Thekkath 1996].

Petal provides a distributed virtual disk abstraction across many disks located on

multiple servers on a switched local network. The virtual disk abstraction tolerates most

hardware and software failures with the aid of replicas of the stored data and

autonomously balances the load on servers by relocating data. Petal virtual disks are

accessed through a UNIX disk driver using standard block input-output operations, so

they can be used to support most file systems. Petal adds between 10 and 100% to the

latency of disk accesses, but the caching strategy results in read and write throughputs

at least as good as those of the underlying disk drives.

Frangipani server modules run within the operating system kernel. As in xFS, the

responsibility for managing files and associated tasks (including the provision of a filelocking service for clients) is assigned to hosts dynamically, and all machines see a

unified file name space with coherent access (with approximately single-copy

semantics) to shared updatable files. Data is stored in a log-structured and striped format

in the Petal virtual disk store. The use of Petal relieves Frangipani of the need to manage

physical disk space, resulting in a much simpler distributed file system implementation.

Frangipani can emulate the service interfaces of several existing file services, including

NFS and DCE/DFS. Frangipani’s performance is at least as good as that of the Digital

implementation of the UNIX file system.SECTION 12.6 SUMMARY 579

12.6 Summary

The key design issues for distributed file systems are:

• the effective use of client caching to achieve performance equal to or better than

that of local file systems;

• the maintenance of consistency between multiple cached client copies of files

when they are updated;

• recovery after client or server failure;

• high throughput for reading and writing files of all sizes;

• scalability.

Distributed file systems are very heavily employed in organizational computing, and

their performance has been the subject of much tuning. NFS has a simple stateless

protocol, but it has maintained its early position as the dominant distributed file system

technology with the help of some relatively minor enhancements to the protocol, tuned

implementations and high-performance hardware support.

AFS demonstrated the feasibility of a relatively simple architecture using server

state to reduce the cost of maintaining coherent client caches. AFS outperforms NFS in

many situations. Recent advances have employed data striping across multiple disks and

log-structured writing to further improve performance and scalability.

Current state-of-the-art distributed file systems are highly scalable, provide good

performance across both local and wide-area networks, maintain one-copy file update

semantics and tolerate and recover from failures. Future requirements include support

for mobile users with disconnected operation, and automatic reintegration and quality of

service guarantees to meet the need for the persistent storage and delivery of streams of

multimedia and other time-dependent data. Solutions to these requirements are

discussed in Chapters 18 and 20.

EXERCISES

12.1 File systems are responsible for the organization, storage, retrieval, naming, sharing and

protection of files. What are the requirements of distributed file systems?

pages 543, 544

12.2 Give one important goal of Sun Microsystems’ Network File System (NFS).

page 546

12.3 Write a procedure PathLookup(Pathname, Dir) → UFID that implements Lookup for

UNIX-like pathnames based on our model directory service. pages 548–550

12.4 What is a file group? What are the properties of the file group identifier? page 551580 CHAPTER 12 DISTRIBUTED FILE SYSTEMS

12.5 To what extent does Sun NFS deviate from one-copy file update semantics? Construct

a scenario in which two user-level processes sharing a file would operate correctly in a

single UNIX host but would observe inconsistencies when running in different hosts.

page 558

12.6 Sun NFS aims to support heterogeneous distributed systems by the provision of an

operating system–independent file service. What are the key decisions that the

implementer of an NFS server for an operating system other than UNIX would have to

take? What constraints should an underlying filing system obey to be suitable for the

implementation of NFS servers? page 552

12.7 Which services must be included to enable clients to bind to services in a given host by

name? page 552

12.8 Give the uses of the following NFS server operations: (i) read (fh, offset, count) (ii) write

(fh, offset, count, data) (iii) lookup (dirfh, name).

page 554

12.9 Explain why the RPC interface to early implementations of NFS is potentially insecure.

The security loophole has been closed in NFS 3 by the use of encryption. How is the

encryption key kept secret? Is the security of the key adequate? pages 555, 560

12.10 After the timeout of an RPC call to access a file on a hard-mounted file system the NFS

client module does not return control to the user-level process that originated the call.

Why? page 555

12.11 How does server caching help to achieve the adequate performance of NFS

implementations? page 558

12.12 How many lookup calls are needed to resolve a five-part pathname (for example,

/usr/users/jim/code/xyz.c) for a file that is stored on an NFS server? What is the reason

for performing the translation step-by-step? page 556

12.13 Kerberos is an authentication system used to secure intranet servers against

unauthorized access and imposter attacks. In what way is the ‘Kerberization’ of NFS

carried out? page 560

12.14 What are the design characteristics of AFS? Name the two software components, which

exist as UNIX processes, with which ANS is implemented. pages 564, 566

12.15 Explain the uses of the following components of the Vice service interface:

(i) SetLock(fid, mode) (ii) RemoveCallback(fid) page 570

12.16 How does AFS deal with the risk that callback messages may be lost? page 568

12.17 Which features of the AFS design make it more scalable than NFS? What are the limits

on its scalability, assuming that servers can be added as required? Which recent

developments offer greater scalability? pages 561, 572, 577581

13

NAME SERVICES

13.1 Introduction

13.2 Name services and the Domain Name System

13.3 Directory services

13.4 Case study: The Global Name Service

13.5 Case study: The X.500 Directory Service

13.6 Summary

This chapter introduces the name service as a distinct service that is used by client

processes to obtain attributes such as the addresses of resources or objects when given

their names. The entities named can be of many types, and they may be managed by

different services. For example, name services are often used to hold the addresses and

other details of users, computers, network domains, services and remote objects. As well

as name services, we describe directory services, which look up services when given

some of their attributes.

Basic design issues for name services, such as the structure and management of

the space of names recognized by the service and the operations that the name service

supports, are outlined and illustrated in the context of the Internet Domain Name

System(DNS).

We also examine how name services are implemented, covering such aspects as

navigation through a collection of name servers when resolving a name, caching naming

data and replicating naming data to increase performance and availability.

Two further case studies are included: the Global Name Service (GNS), and the

X.500 Directory Service, including LDAP.582 CHAPTER 13 NAME SERVICES

13.1 Introduction

In a distributed system, names are used to refer to a wide variety of resources such as

computers, services, remote objects and files, as well as to users. Naming is an issue that

is easily overlooked but is nonetheless fundamental in distributed system design. Names

facilitate communication and resource sharing. A name is needed to request a computer

system to act upon a specific resource chosen out of many; for example, a name in the

form of a URL is needed to access a specific web page. Processes cannot share particular

resources managed by a computer system unless they can name them consistently. Users

cannot communicate with one another via a distributed system unless they can name one

another, for example, with email addresses.

Names are not the only useful means of identification: descriptive attributes are

another. Sometimes clients do not know the name of the particular entity that they seek,

but they do have some information that describes it. Or they may require a service and

know some of its characteristics but not what entity implements it.

This chapter introduces name services, which provide clients with data about

named objects in distributed systems, and the related concept of directory services,

which provide data about objects that satisfy a given description. We describe

approaches to be taken in the design and implementation of these services, using the

Domain Name Service (DNS), the Global Name Service (GNS) and X500 as case

studies. We begin by examining the fundamental concepts of names and attributes.

13.1.1 Names, addresses and other attributes

Any process that requires access to a specific resource must possess a name or an

identifier for it. Examples of human-readable names are file names such as /etc/passwd,

URLs such as http://www.cdk5.net/ and Internet domain names such as www.cdk5.net.

The term identifier is sometimes used to refer to names that are interpreted only by

programs. Remote object references and NFS file handles are examples of identifiers.

Identifiers are chosen for the efficiency with which they can be looked up and stored by

software.

Needham [1993] makes the distinction between a pure name and other names.

Pure names are simply uninterpreted bit patterns. Non-pure names contain information

about the object that they name; in particular, they may contain information about the

location of the object. Pure names always have to be looked up before they can be of any

use. At the other extreme from a pure name is an object’s address: a value that identifies

the location of the object rather than the object itself. Addresses are efficient for

accessing objects, but objects can sometimes be relocated, so addresses are inadequate

as a means of identification. For example, users’ email addresses usually have to change

when they move between organizations or Internet service providers; they are not in

themselves guaranteed to refer to a specific individual over time.

We say that a name is resolved when it is translated into data about the named

resource or object, often in order to invoke an action upon it. The association between a

name and an object is called a binding. In general, names are bound to attributes of the

named objects, rather than the implementation of the objects themselves. An attribute isSECTION 13.1 INTRODUCTION 583

the value of a property associated with an object. A key attribute of an entity that is

usually relevant in a distributed system is its address. For example:

• The DNS maps domain names to the attributes of a host computer: its IP address,

the type of entry (for example, a reference to a mail server or another host) and,

for example, the length of time the host’s entry will remain valid.

• The X500 directory service can be used to map a person’s name onto attributes

including their email address and telephone number.

• The CORBA Naming Service and Trading Service were presented in Chapter 8.

The Naming Service maps the name of a remote object onto its remote object

reference, whereas the Trading Service maps the name of a remote object onto its

remote object reference, together with an arbitrary number of attributes describing

the object in terms understandable by human users.

Note that an ‘address’ may be considered just another name that must be looked up, or

it may contain such a name. An IP address must be looked up to obtain a network

address such as an Ethernet address. Similarly, web browsers and email clients make use

of the DNS to interpret the domain names in URLs and email addresses. Figure 13.1

shows the domain name portion of a URL resolved first via the DNS into an IP address

and then, at the final hop of Internet routing, via ARP to an Ethernet address for the web

server. The last part of the URL is resolved by the file system on the web server to locate

the relevant file.

Names and services • Many of the names used in a distributed system are specific to

some particular service. For example, users of the social networking web site

twitter.com, have names such as @magmapoetry that no other service resolves. Also, a

client may use a service-specific name when requesting a service to perform an

operation upon a named object or resource that it manages. For example, a file name is

given to the file service when requesting that the file be deleted, and a process identifier

is presented to the process management service when requesting that it be sent a signal.

Figure 13.1 Composed naming domains used to access a resource from a URL

http://www.cdk5.net:8888/WebExamples/earth.html

URL

Resource ID (IP number, port number, pathname)

Network address

2:60:8c:2:b0:5a

Web server

55.55.55.55 8888 WebExamples/earth.html

DNS lookup

Socket

file584 CHAPTER 13 NAME SERVICES

These names are used only in the context of the service that manages the objects named,

except when clients communicate about shared objects.

Names are also sometimes needed to refer to entities in a distributed system that

are beyond the scope of any single service. The major examples of these entities are

users (with proper names and email addresses), computers (with hostnames such as

www.cdk5.net) and services themselves (such as file service or printer service). In

object-based middleware, names refer to remote objects that provide services or

applications. Note that many of these names must be readable by and meaningful to

humans, since users and system administrators need to refer to the major components

and configuration of distributed systems, programmers need to refer to services in

programs, and users need to communicate with each other via the distributed system and

discuss what services are available in different parts of it. Given the connectivity

provided by the Internet, these naming requirements are potentially world-wide in

scope.

Uniform Resource Identifiers • Uniform Resource Identifiers (URIs) [Berners-Lee et al.

2005] came about from the need to identify resources on the Web, and other Internet

resources such as electronic mailboxes. An important goal was to identify resources in

a coherent way, so that they could all be processed by common software such as

browsers. URIs are ‘uniform’ in that their syntax incorporates that of indefinitely many

individual types of resource identifiers (that is, URI schemes), and there are procedures

for managing the global namespace of schemes. The advantage of uniformity is that it

eases the process of introducing new types of identifier, as well as using existing types

of identifier in new contexts, without disrupting existing usage.

For example, if someone was to invent a new type of ‘widget’ URI, then URIs

beginning widget: would have to obey the global URI syntax, as well as any local rules

defined for the widget identifier scheme. These URIs would identify widget resources

in a well-defined way. But even existing software that did not access widget resources

could still process widget URIs – for example, by managing directories containing them.

Turning to an example of incorporating existing identifiers, that has been done for

telephone numbers by prefixing them with the scheme name tel and standardizing their

representation, as in tel:+1-816-555-1212. These tel URIs are intended for uses such as

web links that cause telephone calls to be made when invoked.

Uniform Resource Locators: Some URIs contain information that can be used to locate

and access a resource; others are pure resource names. The familiar term Uniform

Resource Locator (URL) is often used for URIs that provide location information and

specify the method for accessing the resource, including the ‘http’ URLs introduced in

Section 1.6. For example, http://www.cdk5.net/ identifies a web page at the given path

(‘/’) on the host www.cdk5.net, and specifies that the HTTP protocol be used to access

it. Another example is a ‘mailto’ URL, such as mailto:fred@flintstone.org, which

identifies the mailbox at the given address.

URLs are efficient identifiers for accessing resources. But they suffer from the

disadvantage that if a resource is deleted or if it moves, say from one web site to another,

there may be dangling links to the resource containing the old URL. If a user clicks on

a dangling link to a web resource, then the web server will either respond that the

resource is not found or – worse, perhaps – supply a different resource that now occupies

the same location.SECTION 13.2 NAME SERVICES AND THE DOMAIN NAME SYSTEM 585

Uniform Resource Names: Uniform Resource Names (URNs) are URIs that are used as

pure resource names rather than locators. For example, the URI:

mid:0E4FC272-5C02-11D9-B115-000A95B55BC8@hpl.hp.com

is a URN that identifies the email message containing it in its ‘Message-Id’ field. The

URI distinguishes that message from any other email message. But it does not provide

the message’s address in any store, so a lookup operation is needed to find it.

A special subtree of URIs beginning with urn: has been reserved for URNs –

although, as the mid: example shows, not all URNs are urn: URIs. The latter urnprefixed URIs are all of the form urn:nameSpace:nameSpace-specificName. For

example, urn:ISBN:0-201-62433-8 identifies books that bear the name 0-201-62433-8 in

the standard ISBN naming scheme. For another example, the (invented) name

urn:doi:10.555/music-pop-1234 refers to the publication called music-pop-1234 in the

naming scheme of the publisher known as 10.555 in the Digital Object Identifier (DOI)

scheme [www.doi.org].

There are resolution services (name services, in the terminology of this chapter)

such as the Handle System [www.handle.net] for resolving URNs such as DOIs to

resource attributes, but none is in widespread use. Indeed, there continues to be debate

in the Web and Internet research communities about the extent to which a separate

category of URNs is needed. One school of thought is that ‘cool URLs do not change’

– in other words, that everyone should assign URLs to resources with guarantees about

their continuity of reference. Against that point of view is the observation that not

everyone is in a position to make such guarantees, which require the wherewithal to

maintain control of a domain name and administer resources carefully.

13.2 Name services and the Domain Name System

A name service stores information about a collection of textual names, in the form of

bindings between the names and the attributes of the entities they denote, such as users,

computers, services and objects. The collection is often subdivided into one or more

naming contexts: individual subsets of the bindings that are managed as a unit. The

major operation that a name service supports is to resolve a name – that is, to look up

attributes from a given name. We describe the implementation of name resolution in

Section 13.2.2. Operations are also required for creating new bindings, deleting bindings

and listing bound names, and adding and deleting contexts.

Name management is separated from other services largely because of the

openness of distributed systems, which brings the following motivations:

Unification: It is often convenient for resources managed by different services to use

the same naming scheme. URIs are a good example of this.

Integration: It is not always possible to predict the scope of sharing in a distributed

system. It may become necessary to share and therefore name resources that were

created in different administrative domains. Without a common name service, the

administrative domains may use entirely different naming conventions.586 CHAPTER 13 NAME SERVICES

General name service requirements • Name services were originally quite simple, since

they were designed only to meet the need to bind names to addresses in a single

management domain, corresponding to a single LAN or WAN. The interconnection of

networks and the increased scale of distributed systems have produced a much larger

name-mapping problem.

Grapevine [Birrell et al. 1982] was one of the earliest extensible, multi-domain

name services. It was designed to be scalable in the number of names and the load of

requests that it could handle.

The Global Name Service, developed at the Digital Equipment Corporation

Systems Research Center [Lampson 1986], is a descendant of Grapevine with ambitious

goals, including:

To handle an essentially arbitrary number of names and to serve an arbitrary number

of administrative organizations: For example, the system should be capable of

handling the names of all the documents in the world.

A long lifetime: Many changes will occur in the organization of the set of names and

in the components that implement the service during its lifetime.

High availability: Most other systems depend upon the name service; they can’t

work when it is broken.

Fault isolation: Local failures should not cause the entire service to fail.

Tolerance of mistrust: A large open system cannot have any component that is

trusted by all of the clients in the system.

Two examples of name services that have concentrated on the goal of scalability to large

numbers of objects such as documents are the Globe name service [van Steen et al.

1998] and the Handle System [www.handle.net]. Far more familiar is the Internet

Domain Name System (DNS), introduced in Chapter 3, which names computers (and

other entities) across the Internet.

In this section, we discuss the main design issues for name services, giving

examples from the DNS. We follow this with a more detailed case study of the DNS.

13.2.1 Name spaces

A name space is the collection of all valid names recognized by a particular service. The

service will attempt to look up a valid name, even though that name may prove not to

correspond to any object – i.e., to be unbound. Name spaces require a syntactic

definition to separate valid names from invalid names. For example, ‘...’ is not

acceptable as the DNS name of a computer, whereas www.cdk99.net is valid (even

though it is unbound).

Names may have an internal structure that represents their position in a hierarchic

name space such as pathnames in a file system, or in an organizational hierarchy such

as Internet domain names; or they may be chosen from a flat set of numeric or symbolic

identifiers. One important advantage of a hierarchy is that it makes large name spaces

more manageable. Each part of a hierarchic name is resolved relative to a separate

context of relatively small size, and the same name may be used with different meanings

in different contexts, to suit different situations of use. In the case of file systems, eachSECTION 13.2 NAME SERVICES AND THE DOMAIN NAME SYSTEM 587

directory represents a context. Thus /etc/passwd is a hierarchic name with two

components. The first, ‘etc’, is resolved relative to the context ‘/’, or root, and the second

part, ‘passwd’, is relative to the context ‘/etc’. The name /oldetc/passwd can have a

different meaning because its second component is resolved in a different context.

Similarly, the same name /etc/passwd may resolve to different files in the contexts of

two different computers.

Hierarchic name spaces are potentially infinite, so they enable a system to grow

indefinitely. By contrast, flat name spaces are usually finite; their size is determined by

fixing a maximum permissible length for names. Another potential advantage of a

hierarchic name space is that different contexts can be managed by different people or

organizations.

The structure of ‘http’ URLs was introduced in Chapter 1. The URL name space

also includes relative names such as ../images/figure1.jpg. When a browser or other web

client encounters such a relative name, it uses the resource in which the relative name is

embedded to determine the server host name and the directory to which this pathname

refers.

DNS names are strings called domain names. Some examples are www.cdk5.net

(a computer), net, com and ac.uk (the latter three are domains).

The DNS name space has a hierarchic structure: a domain name consists of one or

more strings called name components or labels, separated by the delimiter ‘.’. There is

no delimiter at the beginning or end of a domain name, although the root of the DNS

name space is sometimes referred to as ‘.’ for administrative purposes. The name

components are non-null printable strings that do not contain ‘.’. In general, a prefix of

a name is an initial section of the name that contains only zero or more entire

components. For example, in DNS www and www.cdk5 are both prefixes of

www.cdk5.net. DNS names are not case-sensitive, so www.cdk5.net and

WWW.CDK5.NET have the same meaning.

DNS servers do not recognize relative names: all names are referred to the global

root. However, in practical implementations, client software keeps a list of domain

names that are appended automatically to any single-component name before resolution.

For example, the name www presented in the domain cdk5.net probably refers to

www.cdk5.net; client software will append the default domain cdk5.net and attempt to

resolve this name. If this fails, then further default domain names may be appended;

finally, the (absolute) name www will be presented to the root for resolution (an

operation that will of course fail in this case). Names with more than one component,

however, are normally presented intact to the DNS, as absolute names.

Aliases • An alias is a name defined to denote the same information as another name,

similar to a symbolic link between file path names. Aliases allow more convenient

names to be substituted for relatively complicated ones, and allow alternative names to

be used by different people for the same entity. An example is the common use of URL

shorteners, often used in Twitter posts and other situations where space is at a premium.

For example, using web redirection, http://bit.ly/ctqjvH refers to

http://cdk5.net/additional/rmi/programCode/ShapeListClient.java. As another

example, the DNS allows aliases in which one domain name is defined to stand for

another. Aliases are often used to specify the names of machines that run a web server

or an FTP server. For example, the name www.cdk5.net is an alias for cdk5.net. This has588 CHAPTER 13 NAME SERVICES

the advantage that clients can use either name for the web server, and if the web server

is moved to another computer, only the entry for cdk5.net needs to be updated in the

DNS database.

Naming domains • A naming domain is a name space for which there exists a single

overall administrative authority responsible for assigning names within it. This authority

is in overall control of which names may be bound within the domain, but it is free to

delegate this task.

Domains in DNS are collections of domain names; syntactically, a domain’s name

is the common suffix of the domain names within it, but otherwise it cannot be

distinguished from, for example, a computer name. For example, net is a domain that

contains cdk5.net. Note that the term ‘domain name’ is potentially confusing, since only

some domain names identify domains (others identify computers).

The administration of domains may be devolved to subdomains. The domain

dcs.qmul.ac.uk – the Department of Computer Science at Queen Mary, University of

London in the UK – can contain any name the department wishes. But the domain name

dcs.qmul.ac.uk itself had to be agreed with the college authorities, who manage the

domain qmul.ac.uk. Similarly, qmul.ac.uk had to be agreed with the registered authority

for ac.uk, and so on.

Responsibility for a naming domain normally goes hand in hand with

responsibility for managing and keeping up-to-date the corresponding part of the

database stored in an authoritative name server and used by the name service. Naming

data belonging to different naming domains are in general stored by distinct name

servers managed by the corresponding authorities.

Combining and customizing name spaces • The DNS provides a global and homogeneous name space in which a given name refers to the same entity, no matter which process

on which computer looks up the name. By contrast, some name services allow distinct

name spaces – sometimes heterogeneous name spaces – to be embedded into them; and

some name services allow the name space to be customized to suit the needs of individual groups, users or even processes.

Merging: The practice of mounting file systems in UNIX and NFS (see Section 12.3)

provides an example in which a part of one name space is conveniently embedded in

another. But consider how to merge the entire UNIX file systems of two (or more)

computers called red and blue. Each computer has its own root, with overlapping file

names. For example, /etc/passwd refers to one file on red and a different file on blue.

The obvious way to merge the file systems is to replace each computer’s root with a

‘super root’ and mount each computer’s file system in this super root, say as /red and

/blue. Users and programs can then refer to /red/etc/passwd and /blue/etc/passwd. But

the new naming convention by itself would cause programs on the two computers that

still use the old name /etc/passwd to malfunction. A solution is to leave the old root

contents on each computer and embed the mounted file systems /red and /blue of both

computers (assuming that this does not produce name clashes with the old root

contents).

The moral is that we can always merge name spaces by creating a higher-level root

context, but this may raise a problem of backward-compatibility. Fixing the

compatibility problem, in turn, leaves us with hybrid name spaces and the inconvenience

of having to translate old names between the users of the two computers.SECTION 13.2 NAME SERVICES AND THE DOMAIN NAME SYSTEM 589

Heterogeneity: The Distributed Computing Environment (DCE) name space [OSF

1997] allows heterogeneous name spaces to be embedded within it. DCE names may

contain junctions, which are similar to mount points in NFS and UNIX (see Section

12.3), except that they allow heterogeneous name spaces to be mounted. For example,

consider the full DCE name /.../dcs.qmul.ac.uk/principals/Jean.Dollimore. The first part

of this name, /.../dcs.qmul.ac.uk, denotes a context called a cell. The next component is

a junction. For example, the junction principals is a context containing security

principals in which the final component, Jean.Dollimore, may be looked up, and in

which these principal names have their own syntax. Similarly, in

/.../dcs.qmul.ac.uk/files/pub/reports/TR2000-99, the junction files is a context

corresponding to a file system directory, in which the final component

pub/reports/TR2000-99 is looked up and in which the file name space has a distinct

syntax. The two junctions principals and files are the roots of heterogeneous name

spaces, implemented by heterogeneous name services.

Customization: We saw in the example of embedding NFS-mounted file systems above

that sometimes users prefer to construct their name spaces independently rather than

sharing a single name space. File system mounting enables users to import files that are

stored on servers and shared, while the other names continue to refer to local, unshared

files and can be administered autonomously. But the same files accessed from two

different computers may be mounted at different points and thus have different names.

Not sharing the entire name space means users must translate names between

computers.

The Spring naming service [Radia et al. 1993] provides the ability to construct

name spaces dynamically and to share individual naming contexts selectively. Even two

different processes on the same computer can have different naming contexts. Spring

naming contexts are first-class objects that can be shared around a distributed system.

For example, suppose a user on computer red wishes to run a program on blue that issues

file pathnames such as /etc/passwd, but these names are to resolve to the files on red’s

file system, not blue’s. This can be achieved in Spring by passing a reference to red’s

local naming context to blue and using it as the program’s naming context. Plan 9 [Pike

et al. 1993] also allows processes to have their own file system name space. A novel

feature of Plan 9 (which can also be implemented in Spring) is that physical directories

can be ordered and merged into a single logical directory. The effect is that a name

looked up in the single logical directory is looked up in the succession of physical

directories until there is a match, when the attributes are returned. This eliminates the

need to supply lists of paths when looking for program or library files.

13.2.2 Name resolution

For the common case of hierarchic name spaces, name resolution is an iterative or

recursive process whereby a name is repeatedly presented to naming contexts in order

to look up the attributes to which it refers. A naming context either maps a given name

onto a set of primitive attributes (such as those of a user) directly, or maps it onto a

further naming context and a derived name to be presented to that context. To resolve a

name, it is first presented to some initial naming context; resolution iterates as long as

further contexts and derived names are output. We illustrated this at the start of Section590 CHAPTER 13 NAME SERVICES

13.2.1 with the example of /etc/passwd, in which ‘etc’ is presented to the context ‘/’, and

then ‘passwd’ is presented to the context ‘/etc’.

Another example of the iterative nature of resolution is the use of aliases. For

example, whenever a DNS server is asked to resolve an alias such as

www.dcs.qmul.ac.uk, the server first resolves the alias to another domain name (in this

case traffic.dcs.qmul.ac.uk), which must be further resolved to produce an IP address.

In general, the use of aliases makes it possible for cycles to be present in the name

space, in which case resolution may never terminate. Two possible solutions are, to

abandon a resolution process if it passes a threshold number of resolutions, or to leave

administrators to veto any aliases that would introduce cycles.

Name servers and navigation • Any name service, such as DNS, that stores a very large

database and is used by a large population will not store all of its naming information on

a single server computer. Such a server would be a bottleneck and a critical point of

failure. Any heavily used name services should use replication to achieve high

availability. We shall see that DNS specifies that each subset of its database is replicated

in at least two failure-independent servers.

We mentioned above that the data belonging to a naming domain is usually stored

by a local name server managed by the authority responsible for that domain. Although,

in some cases, a name server may store data for more than one domain, it is generally

true to say that data is partitioned into servers according to its domain. We shall see that

in DNS, most of the entries are for local computers. But there are also name servers for

the higher domains, such as yahoo.com and ac.uk, and for the root.

The partitioning of data implies that the local name server cannot answer all

enquiries without the help of other name servers. For example, a name server in the

dcs.qmul.ac.uk domain would not be able to supply the IP address of a computer in the

domain cs.purdue.edu unless it was cached – certainly not the first time it is asked.

The process of locating naming data from more than one name server in order to

resolve a name is called navigation. The client name resolution software carries out

navigation on behalf of the client. It communicates with name servers as necessary to

resolve a name. It may be provided as library code and linked into clients, as for example

in the BIND implementation for DNS (see Section 13.2.3) or in Grapevine [Birrell et al.

1982]. The alternative, used with X500, is to provide name resolution in a separate

process that is shared by all of the client processes on that computer.

One navigation model that DNS supports is known as iterative navigation (see

Figure 13.2). To resolve a name, a client presents the name to the local name server,

which attempts to resolve it. If the local name server has the name, it returns the result

immediately. If it does not, it will suggest another server that will be able to help.

Resolution proceeds at the new server, with further navigation as necessary until the

name is located or is discovered to be unbound.

As DNS is designed to hold entries for millions of domains and is accessed by vast

numbers of clients, it would not be feasible to have all queries starting at a root server,

even if it were replicated heavily. The DNS database is partitioned between servers in

such a way as to allow many queries to be satisfied locally and others to be satisfied

without needing to resolve each part of the name separately. The scheme for resolving

names in DNS is described in more detail in Section 13.2.3.SECTION 13.2 NAME SERVICES AND THE DOMAIN NAME SYSTEM 591

NFS also employs iterative navigation in the resolution of a file name, on a

component-by-component basis (see Chapter 12). This is because the file service may

encounter a symbolic link when resolving a name. A symbolic link must be interpreted

in the client’s file system name space because it may point to a file in a directory stored

at another server. The client computer must determine which server this is, because only

the client knows its mount points.

In multicast navigation, a client multicasts the name to be resolved and the

required object type to the group of name servers. Only the server that holds the named

attributes responds to the request. Unfortunately, however, if the name proves to be

unbound, the request is greeted with silence. Cheriton and Mann [1989] describe a

multicast-based navigation scheme in which a separate server is included in the group

to respond when the required name is unbound.

Another alternative to the iterative navigation model is one in which a name server

coordinates the resolution of the name and passes the result back to the user agent. Ma

[1992] distinguishes non-recursive and recursive server-controlled navigation (Figure

13.3). Under non-recursive server-controlled navigation, any name server may be

chosen by the client. This server communicates by multicast or iteratively with its peers

in the style described above, as though it were a client. Under recursive server-controlled

navigation, the client once more contacts a single server. If this server does not store the

name, the server contacts a peer storing a (larger) prefix of the name, which in turn

attempts to resolve it. This procedure continues recursively until the name is resolved.

If a name service spans distinct administrative domains, then clients executing in

one administrative domain may be prohibited from accessing name servers belonging to

another such domain. Moreover, even name servers may be prohibited from discovering

the disposition of naming data across name servers in another administrative domain.

Then, both client-controlled and non-recursive server-controlled navigation are

inappropriate, and recursive server-controlled navigation must be used. Authorized

name servers request name service data from designated name servers managed by

different administrations, which return the attributes without revealing where the

different parts of the naming database are stored.

Client 1

2 3

Figure 13.2 Iterative navigation

A client iteratively contacts name servers NS1–NS3 in order to resolve a name

Name

servers

NS2

NS3

NS1592 CHAPTER 13 NAME SERVICES

Caching • In DNS and other name services, client name resolution software and servers

maintain a cache of the results of previous name resolutions. When a client requests a

name lookup, the name resolution software consults its cache. If it holds a recent result

from a previous lookup for the name, it returns it to the client; otherwise, it sets about

finding it from a server. That server, in turn, may return data cached from other servers.

Caching is key to a name service’s performance and assists in maintaining the

availability of both the name service and other services in spite of name server crashes.

Its role in enhancing response times by saving communication with name servers is

clear. Caching can be used to eliminate high-level name servers – the root server, in

particular – from the navigation path, allowing resolution to proceed despite some server

failures.

Caching by client name resolvers is widely applied in name services and is

particularly successful because naming data are changed relatively rarely. For example,

information such as computer or service addresses is liable to remain unchanged for

months or years. However, the possibility exists of a name service returning out-of-date

attributes – for example, an out-of-date address – during resolution.

13.2.3 The Domain Name System

The Domain Name System is a name service design whose main naming database is

used across the Internet. It was devised principally by Mockapetris and specified in RFC

1034 [Mockapetris 1987] and RFC 1035. DNS replaced the original Internet naming

scheme, in which all host names and addresses were held in a single central master file

and downloaded by FTP to all computers that required them [Harrenstien et al. 1985].

This original scheme was soon seen to suffer from three major shortcomings:

• It did not scale to large numbers of computers.

• Local organizations wished to administer their own naming systems.

• A general name service was needed – not one that serves only for looking up

computer addresses.

1

2

3

5

1

2

4 3

4

Figure 13.3 Non-recursive and recursive server-controlled navigation

A name server NS1 communicates with other name servers on behalf of a client

client client

Recursive

server-controlled

Non-recursive

server-controlled

NS2

NS1

NS3

NS2

NS1

NS3SECTION 13.2 NAME SERVICES AND THE DOMAIN NAME SYSTEM 593

The objects named by the DNS are primarily computers – for which mainly IP addresses

are stored as attributes – and what we have referred to in this chapter as naming domains

are called simply domains in the DNS. In principle, however, any type of object can be

named, and its architecture gives scope for a variety of implementations. Organizations

and departments within them can manage their own naming data. Millions of names are

bound by the Internet DNS, and lookups are made against it from around the world. Any

name can be resolved by any client. This is achieved by hierarchical partitioning of the

name database, by replication of the naming data, and by caching.

Domain names • The DNS is designed for use in multiple implementations, each of

which may have its own name space. In practice, however, only one is in widespread

use, and that is the one used for naming across the Internet. The Internet DNS name

space is partitioned both organizationally and according to geography. The names are

written with the highest-level domain on the right. The original top-level organizational

domains (also called generic domains) in use across the Internet were:

New top-level domains such as biz and mobi have been added since the early 2000s. A

full list of current generic domain names is available from the Internet Assigned

Numbers Authority [www.iana.org I].

In addition, every country has its own domains:

Countries, particularly those other than the US often use their own subdomains to

distinguish their organizations. The UK, for example, has domains co.uk and ac.uk,

which correspond to com and edu respectively (ac stands for ‘academic community’).

Note that, despite its geographic-sounding uk suffix, a domain such as doit.co.uk

could have data referring to computers in the Spanish office of Doit Ltd., a notional

British company. In other words, even geographic-sounding domain names are

conventional and are completely independent of their physical locations.

DNS queries • The Internet DNS is primarily used for simple host name resolution

and for looking up electronic mail hosts, as follows:

Host name resolution: In general, applications use the DNS to resolve host names

into IP addresses. For example, when a web browser is given a URL containing the

com – Commercial organizations

edu – Universities and other educational institutions

gov – US governmental agencies

mil – US military organizations

net – Major network support centres

org – Organizations not mentioned above

int – International organizations

us – United States

uk – United Kingdom

fr – France

... – ...594 CHAPTER 13 NAME SERVICES

domain name www.dcs.qmul.ac.uk, it makes a DNS enquiry and obtains the

corresponding IP address. As was pointed out in Chapter 4, browsers then use HTTP

to communicate with the web server at the given IP address, using a reserved port

number if none is specified in the URL. FTP and SMTP services work in a similar

way; for example, an FTP program may be given the domain name ftp.dcs.qmul.ac.uk

and can make a DNS enquiry to get its IP address and then use TCP to communicate

with it at the reserved port number.

Mail host location: Electronic mail software uses the DNS to resolve domain names

into the IP addresses of mail hosts – i.e., computers that will accept mail for those

domains. For example, when the address tom@dcs.rnx.ac.uk is to be resolved, the

DNS is queried with the address dcs.rnx.ac.uk and the type designation ‘mail’. It

returns a list of domain names of hosts that can accept mail for dcs.rnx.ac.uk, if such

exist (and, optionally, the corresponding IP addresses). The DNS may return more

than one domain name so that the mail software can try alternatives if the main mail

host is unreachable for some reason. The DNS returns an integer preference value for

each mail host, indicating the order in which the mail hosts should be tried.

Some other types of query that are implemented in some installations but are less

frequently used than those just given are:

Reverse resolution: Some software requires a domain name to be returned given an

IP address. This is just the reverse of the normal host name query, but the name server

receiving the query replies only if the IP address is in its own domain.

Host information: The DNS can store the machine architecture type and operating

system with the domain names of hosts. It has been suggested that this option should

not be used in public, because it provides useful information for those attempting to

gain unauthorized access to computers.

In principle, the DNS can be used to store arbitrary attributes. A query is specified by a

domain name, class and type. For domain names in the Internet, the class is IN. The type

of query specifies whether an IP address, a mail host, a name server or some other type

of information is required. A special domain, in-addr.arpa, exists to hold IP addresses for

reverse lookups. The class attribute is used to distinguish, for example, the Internet

naming database from other (experimental) DNS naming databases. A set of types is

defined for a given database; those for the Internet database are given in Figure 13.5.

DNS name servers • The problems of scale are treated by a combination of partitioning

the naming database and replicating and caching parts of it close to the points of need.

The DNS database is distributed across a logical network of servers. Each server holds

part of the naming database – primarily data for the local domain. Queries concerning

computers in the local domain are satisfied by servers within that domain. However,

each server records the domain names and addresses of other name servers, so that

queries pertaining to objects outside the domain can be satisfied.

The DNS naming data are divided into zones. A zone contains the following data:

• Attribute data for names in a domain, less any subdomains administered by lowerlevel authorities. For example, a zone could contain data for Queen Mary,

University of London – qmul.ac.uk – less the data held by departments (for

example the Department of Computer Science – dcs.qmul.ac.uk).SECTION 13.2 NAME SERVICES AND THE DOMAIN NAME SYSTEM 595

• The names and addresses of at least two name servers that provide authoritative

data for the zone. These are versions of zone data that can be relied upon as being

reasonably up-to-date.

• The names of name servers that hold authoritative data for delegated subdomains;

and ‘glue’ data giving the IP addresses of these servers.

• Zone-management parameters, such as those governing the caching and

replication of zone data.

A server may hold authoritative data for zero or more zones. So that naming data are

available even when a single server fails, the DNS architecture specifies that each zone

must be replicated authoritatively in at least two servers.

System administrators enter the data for a zone into a master file, which is the

source of authoritative data for the zone. There are two types of server that are

considered to provide authoritative data. A primary or master server reads zone data

directly from a local master file. Secondary servers download zone data from a primary

server. They communicate periodically with the primary server to check whether their

stored version matches that held by the primary server. If a secondary’s copy is out of

date, the primary sends it the latest version. The frequency of the secondary’s check is

set by administrators as a zone parameter, and its value is typically once or twice a day.

Any server is free to cache data from other servers to avoid having to contact them

when name resolution requires the same data again; it does this on the proviso that

clients are told that such data is non-authoritative as supplied. Each entry in a zone has

a time-to-live value. When a non-authoritative server caches data from an authoritative

server, it notes the time to live. It will only provide its cached data to clients for up to

this time; when queried after the time period has expired, it recontacts the authoritative

server to check its data. This is a useful feature that minimizes the amount of network

traffic while retaining flexibility for system administrators. When attributes are

expected to change rarely, they can be given a correspondingly large time to live. If an

administrator knows that attributes are likely to change soon, they can reduce the time

to live accordingly.

Figure 13.4 shows the arrangement of some of the DNS database as it stood in the

year 2001. This example is equally valid today even if some of the data has altered as

systems have been reconfigured over time. Note that, in practice, root servers such as

a.root-servers.net hold entries for several levels of domain, as well as entries for firstlevel domain names. This is to reduce the number of navigation steps required to resolve

domain names. Root name servers hold authoritative entries for the name servers for the

top-level domains. They are also authoritative name servers for the generic top-level

domains, such as com and edu. However, the root name servers are not name servers for

the country domains. For example, the uk domain has a collection of name servers, one

of which is called ns1.nic.net. These name servers know the name servers for the

second-level domains in the United Kingdom such as ac.uk and co.uk. The name servers

for the domain ac.uk know the name servers for all of the university domains in the

country, such as qmul.ac.uk or ic.ac.uk. In some cases, a university domain delegates

some of its responsibilities to a subdomain, such as dcs.qmul.ac.uk.

The root domain information is replicated by a primary server to a collection of

secondary servers, as described above. In spite of this, root servers serve thousands or596 CHAPTER 13 NAME SERVICES

more queries per second. All DNS servers store the addresses of one or more root name

servers, which do not change very often. They also usually store the address of an

authoritative server for the parent domain. A query involving a three-component domain

name such as www.berkeley.edu can be satisfied using at worst two navigation steps:

one to a root server that stores an appropriate name server entry, and a second to the

server whose name is returned.

Referring to Figure 13.4, the domain name jeans-pc.dcs.qmul.ac.uk can be looked

up from within dcs.qmul.ac.uk using the local server dns0.dcs.qmul.ac.uk. This server

does not store an entry for the web server www.ic.ac.uk, but it does keep a cached entry

for ic.ac.uk (which it obtained from the authorized server ns0.ja.net). The server dns0-

doc.ic.ac.uk can be contacted to resolve the full name.

Navigation and query processing • A DNS client is called a resolver. It is normally

implemented as library software. It accepts queries, formats them into messages in the

form expected under the DNS protocol and communicates with one or more name

servers in order to satisfy the queries. A simple request-reply protocol is used, typically

using UDP packets on the Internet (DNS servers use a well-known port number). The

resolver times out and resends its query if necessary. The resolver can be configured to

contact a list of initial name servers in order of preference in case one or more are

unavailable.

Figure 13.4 DNS name servers

a.root-servers.net

(root)

ns0.ja.net

(ac.uk)

dns0.dcs.qmul.ac.uk

(dcs.qmul.ac.uk)

alpha.qmul.ac.uk

(qmul.ac.uk)

dns0-doc.ic.ac.uk

(ic.ac.uk)

ns.purdue.edu

(purdue.edu)

uk

purdue.edu

ic.ac.uk

qmul.ac.uk

dcs.qmul.ac.uk

\*.qmul.ac.uk

\*.dcs.qmul.ac.uk \*.ic.ac.uk

\*.purdue.edu

Name server names are in italics, and the corresponding domains are in

parentheses. Arrows denote name server entries

ns1.nic.uk

(uk)

ac.uk

co.uk

yahoo.comSECTION 13.2 NAME SERVICES AND THE DOMAIN NAME SYSTEM 597

The DNS architecture allows for recursive navigation as well as iterative

navigation. The resolver specifies which type of navigation is required when contacting

a name server. However, name servers are not bound to implement recursive navigation.

As was pointed out above, recursive navigation may tie up server threads, meaning that

other requests might be delayed.

In order to save on network communication, the DNS protocol allows for multiple

queries to be packed into the same request message and for name servers

correspondingly to send multiple replies in their response messages.

Resource records • Zone data are stored by name servers in files in one of several fixed

types of resource record. For the Internet database, these include the types given in

Figure 13.5. Each record refers to a domain name, which is not shown. The entries in the

table refer to items already mentioned, except that AAAA records store IPv6 addresses

whereas A records store IPv4 addresses, and TXT entries are included to allow arbitrary

other information to be stored along with domain names.

The data for a zone starts with an SOA-type record, which contains the zone

parameters that specify, for example, the version number and how often secondaries

should refresh their copies. This is followed by a list of records of type NS specifying

the name servers for the domain and a list of records of type MX giving the domain

names of mail hosts, each prefixed by a number expressing its preference. For example,

part of the database for the domain dcs.qmul.ac.uk at one point is shown in Figure 13.6,

where the time to live 1D means 1 day.

Further records of type A later in the database give the IP addresses for the two

name servers dns0 and dns1. The IP addresses of the mail hosts and the third name server

are given in the databases corresponding to their domains.

Figure 13.5 DNS resource records

Record type Meaning Main contents

A A computer address (IPv4) IPv4 number

AAAA A computer address (IPv6) IPv6 number

NS An authoritative name server Domain name for server

CNAME The canonical name for an alias Domain name for alias

SOA Marks the start of data for a zone Parameters governing the zone

PTR Domain name pointer (reverse

lookups) Domain name

HINFO Host information Machine architecture and operating

system

MX Mail exchange List of <preference, host> pairs

TXT Text string Arbitrary text598 CHAPTER 13 NAME SERVICES

The majority of the remainder of the records in a lower-level zone like

dcs.qmul.ac.uk will be of type A and map the domain name of a computer onto its IP

address. They may contain some aliases for the well-known services, for example:

If the domain has any subdomains, there will be further records of type NS specifying

their name servers, which will also have individual A entries. For example, at one point

the database for qmul.ac.uk contained the following records for the name servers in its

subdomain dcs.qmul.ac.uk:

Load sharing by name servers: At some sites, heavily used services such as the Web and

FTP are supported by a group of computers on the same network. In this case, the same

domain name is used for each member of the group. When a domain name is shared by

several computers, there is one record for each computer in the group, giving its IP

address. By default, the name server responds to queries for which multiple records

match the requested name by returning the IP addresses according to a round-robin

schedule. Successive clients are given access to different servers so that the servers can

share the workload. Caching has a potential for spoiling this scheme, for once a nonauthoritative name server or a client has the server’s address in its cache it will continue

to use it. To counteract this effect, the records are given a short time to live.

The BIND implementation of the DNS • The Berkeley Internet Name Domain (BIND) is

an implementation of the DNS for computers running UNIX. Client programs link in

library software as the resolver. DNS name server computers run the named daemon.

domain name time to live class type value

www 1D IN CNAME traffic

traffic 1D IN A 138.37.95.150

domain name time to live class type value

dcs 1D IN NS dns0.dcs

dns0.dcs 1D IN A 138.37.88.249

dcs 1D IN NS dns1.dcs

dns1.dcs 1D IN A 138.37.94.248

Figure 13.6 DNS zone data records

domain name time to live class type value

dcs.qmul.ac.uk 1D IN NS dns0

dcs.qmul.ac.uk 1D IN NS dns1

dcs.qmul.ac.uk 1D IN MX 1 mail1.qmul.ac.uk

dcs.qmul.ac.uk 1D IN MX 2 mail2.qmul.ac.ukSECTION 13.2 NAME SERVICES AND THE DOMAIN NAME SYSTEM 599

BIND allows for three categories of name server: primary servers, secondary

servers and caching-only servers. The named program implements just one of these

types, according to the contents of a configuration file. The first two categories are as

described above. Caching-only servers read in from a configuration file sufficient names

and addresses of authoritative servers to resolve any name. Thereafter, they only store

this data and data that they learn by resolving names for clients.

A typical organization has one primary server, with one or more secondary servers

that provide name serving on different local area networks at the site. Additionally,

individual computers often run their own caching-only server, to reduce network traffic

and speed up response times still further.

Discussion of the DNS • The DNS Internet implementation achieves relatively short

average response times for lookups, considering the amount of naming data and the

scale of the networks involved. We have seen that it achieves this by a combination of

partitioning, replicating and caching naming data. The objects named are primarily

computers, name servers and mail hosts. Computer (host) name–to–IP address

mappings change relatively rarely, as do the identities of name servers and mail hosts,

so caching and replication occur in a relatively clement environment.

The DNS allows naming data to become inconsistent. That is, if naming data is

changed, then other servers may provide clients with stale data for periods on the order

of days. None of the replication techniques explored in Chapter 18 is applied. However,

inconsistency is of no consequence until such time as a client attempts to use stale data.

The DNS does not address itself to how staleness of addresses is detected.

Apart from computers, the DNS also names one particular type of service – the

mail service – on a per-domain basis. DNS assumes there to be only one mail service per

addressed domain, so users do not have to include the name of this service explicitly in

names. Electronic mail applications transparently select this service by using the

appropriate type of query when contacting DNS servers.

In summary, the DNS stores a limited variety of naming data, but this is sufficient

in so far as applications such as electronic mail impose their own naming schemes on

top of domain names. It might be argued that the DNS database represents the lowest

common denominator of what would be considered useful by the many user

communities on the Internet. The DNS was not designed to be the only name service in

the Internet; it coexists with local name and directory services that store data most

pertinent to local needs (such as Sun’s Network Information Service, which stores

encoded passwords, for example, or Microsoft’s Active Directory Services

[www.microsoft.com I], which stores detailed information about all the resources within

a domain).

What remains as a potential problem for the DNS design is its rigidity with respect

to changes in the structure of the name space, and the lack of ability to customize the

name space to suit local needs. These aspects of naming design are taken up by the case

study of the Global Name Service in Section 13.4. But before that, we consider directory

services.600 CHAPTER 13 NAME SERVICES

13.3 Directory services

We have described how name services store collections of <name, attribute> pairs, and

how the attributes are looked up from a name. It is natural to consider the dual of this

arrangement, in which attributes are used as values to be looked up. In these services,

textual names can be considered to be just another attribute. Sometimes users wish to

find a particular person or resource, but they do not know its name, only some of its other

attributes. For example, a user may ask: ‘What is the name of the user with telephone

number 020-555 9980?’ Likewise, sometimes users require a service, but they are not

concerned with what system entity supplies that service, as long as the service is

conveniently accessible. For example, a user might ask, ‘Which computers in this

building are Macintoshes running the Mac OS X operating system?’ or ‘Where can I

print a high-resolution colour image?’

A service that stores collections of bindings between names and attributes and that

looks up entries that match attribute-based specifications is called a directory service.

Examples are Microsoft’s Active Directory Services, X.500 and its cousin LDAP

(described in Section 13.5), Univers [Bowman et al. 1990] and Profile [Peterson 1988].

Directory services are sometimes called yellow pages services, and conventional name

services are correspondingly called white pages services, in an analogy with the

traditional types of telephone directory. Directory services are also sometimes known as

attribute-based name services.

A directory service returns the sets of attributes of any objects found to match

some specified attributes. So, for example, the request ‘TelephoneNumber = 020 555

9980’ might return {‘Name = John Smith’, ‘TelephoneNumber = 020 555 9980’,

‘emailAddress = john@dcs.gormenghast.ac.uk’, ...}. The client may specify that only a

subset of the attributes is of interest – for example, just the email addresses of matching

objects. X.500 and some other directory services also allow objects to be looked up by

conventional hierarchic textual names. The Universal Directory and Discovery Service

(UDDI), which was presented in Section 9.4, provides both white pages and yellow

pages services to provide information about organizations and the web services they

offer.

UDDI aside, the term discovery service normally denotes the special case of a

directory service for services provided by devices in a spontaneous networking

environment. As Section 1.3.2 described, devices in spontaneous networks are liable to

connect and disconnect unpredictably. One core difference between a discovery service

and other directory services is that the address of a directory service is normally well

known and preconfigured in clients, whereas a device entering a spontaneous

networking environment has to resort to multicast navigation, at least the first time it

accesses the local discovery service. Section 19.2.1 describes discovery services in

detail.

Attributes are clearly more powerful than names as designators of objects:

programs can be written to select objects according to precise attribute specifications

where names might not be known. Another advantage of attributes is that they do not

expose the structure of organizations to the outside world, as do organizationally

partitioned names. However, the relative simplicity of use of textual names makes them

unlikely to be replaced by attribute-based naming in many applications.SECTION 13.4 CASE STUDY: THE GLOBAL NAME SERVICE 601

13.4 Case study: The Global Name Service

A Global Name Service (GNS) was designed and implemented by Lampson and

colleagues at the DEC Systems Research Center [Lampson 1986] to provide facilities

for resource location, mail addressing and authentication. The design goals of the GNS

have already been listed at the end of Section 13.1; they reflect the fact that a name

service for use in an internetwork must support a naming database that may extend to

include the names of millions of computers and (eventually) email addresses for billions

of users. The designers of the GNS also recognized that the naming database is likely to

have a long lifetime and that it must continue to operate effectively while it grows from

small to large scale and while the network on which it is based evolves. The structure of

the name space may change during that time to reflect changes in organizational

structures. The service should accommodate changes in the names of the individuals,

organizations and groups that it holds, and changes in the naming structure such as those

that occur when one company is taken over by another. In this description, we focus on

those features of the design that enable it to accommodate such changes.

The potentially large naming database and the scale of the distributed environment

in which the GNS is intended to operate make the use of caching essential and render it

extremely difficult to maintain complete consistency between all copies of a database

entry. The cache consistency strategy adopted relies on the assumption that updates to

the database will be infrequent and that slow dissemination of updates is acceptable,

since clients can detect and recover from the use of out-of-date naming data.

The GNS manages a naming database that is composed of a tree of directories

holding names and values. Directories are named by multi-part pathnames referred to a

root, or relative to a working directory, much like file names in a UNIX file system. Each

directory is also assigned an integer, which serves as a unique directory identifier (DI).

In this section, we use names in italics when referring to the DI of a directory, so that

EC is the identifier of the EC directory. A directory contains a list of names and

references. The values stored at the leaves of the directory tree are organized into value

trees, so that the attributes associated with names can be structured values.

Names in the GNS have two parts: <directory name, value name>. The first part

identifies a directory; the second refers to a value tree, or some portion of a value tree.

For example, see Figure 13.7, in which the DIs are illustrated as small integers (although

they are actually chosen from a range of integers to ensure uniqueness). The attributes

of a user Peter.Smith in the directory QMUL would be stored in the value tree named

<EC/UK/AC/QMUL, Peter.Smith>. The value tree includes a password, which can be

referenced as <EC/UK/AC/QMUL, Peter.Smith/password>, and several mail addresses,

each of which would be listed in the value tree as a single node with the name

<EC/UK/AC/QMUL, Peter.Smith/mailboxes>.

The directory tree is partitioned and stored in many servers, with each partition

replicated in several servers. The consistency of the tree is maintained in the face of two

or more concurrent updates – for example, two users may simultaneously attempt to

create entries with the same name, and only one should succeed. Replicated directories

present a second consistency problem; this is addressed by an asynchronous update

distribution algorithm that ensures eventual consistency, but with no guarantee that all

copies are always current.602 CHAPTER 13 NAME SERVICES

Accommodating change • We now turn to the aspects of the design that are concerned

with accommodating growth and change in the structure of the naming database. At the

level of clients and administrators, growth is accommodated through extension of the

directory tree in the usual manner. But we may wish to integrate the naming trees of two

previously separate GNS services. For example, how could we integrate the database

rooted at the EC directory shown in Figure 13.7 with another database for NORTH

AMERICA? Figure 13.8 shows a new root, WORLD, introduced above the existing roots

of the two trees to be merged. This is a straightforward technique, but how does it affect

clients that continue to use names that are referred to what was ‘the root’ before

integration took place? For example, </UK/AC/QMUL, Peter.Smith> is a name used by

clients before integration. It is an absolute name (since it begins with the symbol for the

root, ‘/’), but the root it refers to is EC, not WORLD. EC and NORTH AMERICA are

working roots – initial contexts against which names beginning with the root ‘/’ are to

be looked up.

Figure 13.7 GNS directory tree and value tree for user Peter.Smith

UK FR

AC

DI: 322 QMUL

Peter.Smith

mailboxes password

DI: 599 (EC)

DI: 543 DI: 574

DI: 437

Alpha Gamma Beta

Figure 13.8 Merging trees under a new root

EC

UK FR

DI: 599

DI: 543 DI: 574

NORTH AMERICA

US

DI: 642

DI: 732 DI: 457

#599 = #633/EC

#642 = #633/NORTH AMERICA

Well-known directories:

CANADA

DI: 633 (WORLD)SECTION 13.4 CASE STUDY: THE GLOBAL NAME SERVICE 603

The existence of unique directory identifiers can be used to solve this problem.

The working root for each program must be identified as part of its execution

environment (much as is done for a program’s working directory). When a client in the

European Community uses a name of the form </UK/AC/QMUL, Peter.Smith>, its

local user agent, which is aware of the working root, prefixes the directory identifier EC

(#599), thus producing the name <#599/UK/AC/QMUL, Peter.Smith>. The user agent

passes this derived name in the lookup request to a GNS server. The user agent may deal

similarly with relative names referred to working directories. Clients that are aware of

the new configuration may also supply absolute names to the GNS server, which are

referred to the conceptual super-root directory containing all directory identifiers – for

example, <WORLD/EC/UK/AC/QMUL, Peter.Smith> – but the design cannot assume

that all clients will be updated to take account of such a change.

The technique described above solves the logical problem, allowing users and

client programs to continue to use names that are defined relative to an old root even

when a new real root is inserted, but it leaves an implementation problem: in a

distributed naming database that may contain millions of directories, how can the GNS

service locate a directory given only its identifier, such as #599? The solution adopted

by the GNS is to list those directories that are used as working roots, such as EC, in a

table of ‘well-known directories’ held in the current real root directory of the naming

database. Whenever the real root of the naming database changes, as it does in Figure

13.8, all GNS servers are informed of the new location of the real root. They can then

interpret names of the form WORLD/EC/UK/AC/QMUL (referred to the real root) in

the usual way, and they can interpret names of the form #599/UK/AC/QMUL by using

the table of ‘well-known directories’ to translate them to full pathnames beginning at the

real root.

The GNS also supports the restructuring of the database to accommodate

organizational change. Suppose that the United States becomes part of the European

Community (!). Figure 13.9 shows the new directory tree. But if the US subtree is simply

moved to the EC directory, names beginning WORLD/NORTH AMERICA/US will no

longer work. The solution adopted by the GNS is to insert a ‘symbolic link’ in place of

the original US entry (shown in bold in Figure 13.9). The GNS directory lookup

procedure interprets the link as a redirection to the US directory in its new location.

Figure 13.9 Restructuring the directory

EC

UK FR

DI: 599

DI: 543 DI: 574

NORTH AMERICA

US

DI: 642

DI: 732 DI: 457

#599 = #633/EC

#642 = #633/NORTH AMERICA

Well-known directories:

CANADA

DI: 633 (WORLD)

#633/EC/US

US604 CHAPTER 13 NAME SERVICES

Discussion of the GNS • The GNS is descended from Grapevine [Birrell et al. 1982] and

Clearinghouse [Oppen and Dalal 1983], two successful naming systems developed

primarily for the purposes of mail delivery by the Xerox Corporation. The GNS

successfully addresses needs for scalability and reconfigurability, but the solution

adopted for merging and moving directory trees results in a requirement for a database

(the table of well-known directories) that must be replicated at every node. In a largescale network, reconfigurations may occur at any level, and this table could grow to a

large size, conflicting with the scalability goal.

13.5 Case study: The X.500 Directory Service

X.500 is a directory service in the sense defined in Section 13.3. It can be used in the

same way as a conventional name service, but it is primarily used to satisfy descriptive

queries and is designed to discover the names and attributes of other users or system

resources. Users may have a variety of requirements for searching and browsing in a

directory of network users, organizations and system resources to obtain information

about the entities that the directory contains. The uses for such a service are likely to be

quite diverse. They range from enquiries that are directly analogous to the use of

telephone directories, such as a simple ‘white pages’ access to obtain a user’s electronic

mail address or a ‘yellow pages’ query aimed, for example, at obtaining the names and

telephone numbers of garages specializing in the repair of a particular make of car, to

the use of the directory to access personal details such as job roles, dietary habits or even

photographic images of the individuals.

Such queries may originate from users, in the ‘yellow pages’example mentioned

above, or from processes, when they may be used to identify services to meet a

functional requirement.

Individuals and organizations can use a directory service to make available a wide

range of information about themselves and the resources that they wish to offer for use

in the network. Users can search the directory for specific information with only partial

knowledge of its name, structure or content.

The ITU and ISO standards organizations defined the X.500 Directory Service

[ITU/ISO 1997] as a network service intended to meet these requirements. The standard

refers to it as a service for access to information about ‘real-world entities’, but it is also

likely to be used for access to information about hardware and software services and

devices. X.500 is specified as an application-level service in the Open Systems

Interconnection (OSI) set of standards, but its design does not depend to any significant

extent on the other OSI standards, and it can be viewed as a design for a general-purpose

directory service. We outline the design of the X.500 directory service and its

implementation here. Readers interested in a more detailed description of X.500 and

methods for its implementation are advised to study Rose’s book on the subject [Rose

1992]. X.500 is also the basis for LDAP (discussed below), and it is used in the DCE

directory service [OSF 1997].SECTION 13.5 CASE STUDY: THE X.500 DIRECTORY SERVICE 605

The data stored in X.500 servers is organized in a tree structure with named nodes,

as in the case of the other name servers discussed in this chapter, but in X.500 a wide

range of attributes are stored at each node in the tree, and access is possible not just by

name but also by searching for entries with any required combination of attributes.

The X.500 name tree is called the Directory Information Tree (DIT), and the

entire directory structure including the data associated with the nodes, is called the

Directory Information Base (DIB). There is intended to be a single integrated DIB

containing information provided by organizations throughout the world, with portions

of the DIB located in individual X.500 servers. Typically, a medium-sized or large

organization would provide at least one server. Clients access the directory by

establishing a connection to a server and issuing access requests. Clients can contact any

server with an enquiry. If the data required are not in the segment of the DIB held by the

contacted server, it will either invoke other servers to resolve the query or redirect the

client to another server.

In the terminology of the X.500 standard, servers are Directory Service Agents

(DSAs), and their clients are termed Directory User Agents (DUAs). Figure 13.10

shows the software architecture and one of the several possible navigation models, with

each DUA client process interacting with a single DSA process, which accesses other

DSAs as necessary to satisfy requests.

Each entry in the DIB consists of a name and a set of attributes. As in other name

servers, the full name of an entry corresponds to a path through the DIT from the root of

the tree to the entry. In addition to full or absolute names, a DUA can establish a context,

which includes a base node, and then use shorter relative names that give the path from

the base node to the named entry.

Figure 13.11 shows the portion of the Directory Information Tree that includes the

notional University of Gormenghast in Great Britain, and Figure 13.12 is one of the

associated DIB entries. The data structure for the entries in the DIB and the DIT is very

flexible. A DIB entry consists of a set of attributes, where an attribute has a type and one

or more values. The type of each attribute is denoted by a type name (for example,

countryName, organizationName, commonName, telephoneNumber, mailbox,

objectClass). New attribute types can be defined if they are required. For each distinct

Figure 13.10 X.500 service architecture

DSA

DSA

DSA

DSA

DUA DSA DSA

DUA

DUA606 CHAPTER 13 NAME SERVICES

type name there is a corresponding type definition, which includes a type description

and a syntax definition in the ASN.1 notation (a standard notation for syntax definitions)

defining representations for all permissible values of the type.

DIB entries are classified in a manner similar to the object class structures found

in object-oriented programming languages. Each entry includes an objectClass attribute,

which determines the class (or classes) of the object to which an entry refers.

Organization, organizationalPerson and document are all examples of objectClass

values. Further classes can be defined as they are required. The definition of a class

determines which attributes are mandatory and which are optional for entries of the

given class. The definitions of classes are organized in an inheritance hierarchy in which

all classes except one (called topClass) must contain an objectClass attribute, and the

value of the objectClass attribute must be the names of one or more classes. If there are

several objectClass values, the object inherits the mandatory and optional attributes of

each of the classes.

The name of a DIB entry (the name that determines its position in the DIT) is

determined by selecting one or more of its attributes as distinguished attributes. The

attributes selected for this purpose are referred to as the entry’s Distinguished Name

(DN).

Figure 13.11 Part of the X.500 Directory Information Tree

... France (country) Great Britain (country) Greece (country) ...

... BT Plc (organization) University of Gormenghast (organization) ...

Department of Computer Science (organizationalUnit)

Computing Service (organizationalUnit)

Engineering Department (organizationalUnit)

...

...

X.500 Service (root)

Departmental Staff (organizationalUnit)

Research Students (organizationalUnit)

ely (applicationProcess)

...

...

... Alice Flintstone (person) Pat King ... (person) James Healey (person) Janet Papworth (person) ...SECTION 13.5 CASE STUDY: THE X.500 DIRECTORY SERVICE 607

Now we can consider the methods by which the directory is accessed. There are

two main types of access request:

read: An absolute or relative name (a domain name in X.500 terminology) for an

entry is given, together with a list of attributes to be read (or an indication that all

attributes are required). The DSA locates the named entry by navigating in the DIT,

passing requests to other DSA servers where it does not hold relevant parts of the tree.

It retrieves the required attributes and returns them to the client.

search: This is an attribute-based access request. A base name and a filter expression

are supplied as arguments. The base name specifies the node in the DIT from which

the search is to commence; the filter expression is a boolean expression that is to be

evaluated for every node below the base node. The filter specifies a search criterion:

a logical combination of tests on the values of any of the attributes in an entry. The

search command returns a list of names (domain names) for all of the entries below

the base node for which the filter evaluates to TRUE.

For example, a filter might be constructed and applied to find the

commonNames of members of staff who occupy room Z42 in the Department of

Computer Science at the University of Gormenghast (Figure 13.12). A read request

could then be used to obtain any or all of the attributes of those DIB entries.

Searching can be quite costly when it is applied to large portions of the

directory tree (which may reside in several servers). Additional arguments can be

supplied to search to restrict the scope, the time for which the search is allowed to

continue and the size of the list of entries that is returned.

Administration and updating of the DIB • The DSA interface includes operations for

adding, deleting and modifying entries. Access control is provided for both queries and

updating operations, so access to parts of the DIT may be restricted to certain users or

classes of user.

Figure 13.12 An X.500 DIB Entry

info

Alice Flintstone, Departmental Staff, Department of Computer Science,

University of Gormenghast, GB

commonName

Alice.L.Flintstone

Alice.Flintstone

Alice Flintstone

A. Flintstone

surname

Flintstone

telephoneNumber

+44 986 33 4604

uid

alf

mail

alf@dcs.gormenghast.ac.uk

Alice.Flintstone@dcs.gormenghast.ac.uk

roomNumber

Z42

userClass

Research Fellow608 CHAPTER 13 NAME SERVICES

The DIB is partitioned, with the expectation that each organization will provide at

least one server holding the details of the entities in that organization. Portions of the

DIB may be replicated in several servers.

As a standard (or a ‘recommendation’ in CCITT terminology), X.500 does not

address implementation issues. However, it is quite clear that any implementation

involving multiple servers in a wide area network must rely on extensive use of

replication and caching techniques to avoid too much redirection of queries.

One implementation, described by Rose [1992], is a system developed at

University College, London, known as QUIPU [Kille 1991]. In this implementation,

both caching and replication are performed at the level of individual DIB entries, and at

the level of collections of entries descended from the same node. It is assumed that

values may become inconsistent after an update, and the time interval in which the

consistency is restored may be several minutes. This form of update dissemination is

generally considered acceptable for directory service applications.

Lightweight Directory Access Protocol • X.500’s assumption that organizations would

provide information about themselves in public directories within a common system has

proved largely unfounded. Equally, its compexity has meant that its uptake has been

relatively modest.

A group at the University of Michigan proposed a more lightweight approach

called the Lightweight Directory Access Protocol (LDAP), in which a DUA accesses

X.500 directory services directly over TCP/IP instead of the upper layers of the ISO

protocol stack. This is described in RFC 2251 [Wahl et al. 1997]. LDAP also simplifies

the interface to X.500 in other ways: for example, it provides a relatively simple API and

it replaces ASN.1 encoding with textual encoding.

Although the LDAP specification is based on X.500, LDAP does not require it.

An implementation may use any other directory server that obeys the simpler LDAP

specification, as opposed to the X.500 specification. For example, Microsoft’s Active

Directory Services provides an LDAP interface.

Unlike X.500, LDAP has been widely adopted, particularly for intranet directory

services. It provides secure access to directory data through authentication.

13.6 Summary

This chapter has described the design and implementation of name services in

distributed systems. Name services store the attributes of objects in a distributed system

– in particular, their addresses – and return these attributes when a textual name is

supplied to be looked up.

The main requirements for the name service are an ability to handle an arbitrary

number of names, a long lifetime, high availability, the isolation of faults and the

tolerance of mistrust.

The primary design issue is the structure of the name space – the syntactic rules

governing names. A related issue is the resolution model, which sets out the rules by

which a multi-component name is resolved to a set of attributes. The set of bound names

must be managed. Most designs consider the name space to be divided into domains –SECTION 13.6 SUMMARY 609

discrete sections of the name space, each of which is associated with a single authority

controlling the binding of names within it.

The implementation of the name service may span different organizations and

user communities. The collection of bindings between names and attributes, in other

words, is stored at multiple name servers, each of which stores at least part of the set of

names within a naming domain. The question of navigation therefore arises – by what

procedure can a name be resolved when the necessary information is stored at several

sites? The types of navigation that are supported are iterative, multicast, recursive

server-controlled and non-recursive server-controlled.

Another important aspect of the implementation of a name service is the use of

replication and caching. Both of these assist in making the service highly available, and

both also reduce the time taken to resolve a name.

This chapter has considered two main cases of name service design and

implementation. The Domain Name System is widely used for naming computers and

addressing electronic mail across the Internet; it achieves good response times through

replication and caching. The Global Name Service is a design that has tackled the issue

of reconfiguring the name space as organizational changes occur.

The chapter also considered directory services, which provide data about

matching objects and services when clients supply attribute-based descriptions. X.500

is a model for directory services that can range in scope from individual organizations

to global directories. It has been taken up more widely for use in intranets since the

arrival of the LDAP software.

EXERCISES

13.1 Describe the advantages of the uniformity of Uniform Resource Identifiers (URIs) and

Uniform Resource Locators (URLs). page 584

13.2 Discuss the problem associated with name services in a distributed system. How can this

be solved? page 585

13.3 Explain why a name space is important for a particular name service. What is the

advantage of a hierarchic name space?

page 586

13.4 Describe the heterogeneity of the Distributed Computing Environment (DCE) name

space. What is its cell and junction in this context? Give an example. page 589

13.5 Why does NFS employ iterative navigation in the resolution of a file name? page 591

13.6 Discuss the shortcomings of the original Internet naming scheme, in which all host

names and addresses were held in a single central master file. page 592

13.7 Investigate your local configuration of DNS domains and servers. You may find a

program such as dig or nslookup installed, which enables you to carry out individual

name server queries. page 594610 CHAPTER 13 NAME SERVICES

13.8 Why do DNS root servers hold entries for two-level names such as yahoo.com and

purdue.edu, rather than one-level names such as edu and com? page 595

13.9 Which are the original top-level organizational domains in use across the Internet?

page 593

13.10 Why might a DNS client choose recursive navigation rather than iterative navigation?

What is the relevance of the recursive navigation option to concurrency within a name

server? page 597

13.11 A DNS client is called a resolver. What is its role?

page 597

13.12 The GNS does not guarantee that all copies of entries in the naming database are up-todate. How are clients of the GNS likely to become aware that they have been given an

out-of-date entry? Under what circumstances might it be harmful? page 601

13.13 Discuss the potential advantages and drawbacks of the use of an X.500 directory service

in place of the DNS and the Internet mail delivery programs. Sketch the design of a mail

delivery system for an internetwork in which all mail users and mail hosts are registered

in an X.500 database. page 604

13.14 What features does the X.500 directory service provide over a conventional name

service? page 604611

14

TIME AND GLOBAL STATES

14.1 Introduction

14.2 Clocks, events and process states

14.3 Synchronizing physical clocks

14.4 Logical time and logical clocks

14.5 Global states

14.6 Distributed debugging

14.7 Summary

In this chapter, we introduce some topics related to the issue of time in distributed

systems. Time is an important practical issue. For example, we require computers around

the world to timestamp electronic commerce transactions consistently. Time is also an

important theoretical construct in understanding how distributed executions unfold. But

time is problematic in distributed systems. Each computer may have its own physical

clock, but the clocks typically deviate, and we cannot synchronize them perfectly. We

examine algorithms for synchronizing physical clocks approximately and then go on to

explain logical clocks, including vector clocks, which are a tool for ordering events without

knowing precisely when they occurred.

The absence of global physical time makes it difficult to find out the state of our

distributed programs as they execute. We often need to know what state process A is in

when process B is in a certain state, but we cannot rely on physical clocks to know what

is true at the same time. The second half of the chapter examines algorithms to determine

global states of distributed computations despite the lack of global time.612 CHAPTER 14 TIME AND GLOBAL STATES

14.1 Introduction

This chapter introduces fundamental concepts and algorithms related to monitoring

distributed systems as their execution unfolds, and to timing the events that occur in their

executions.

Time is an important and interesting issue in distributed systems, for several

reasons. First, time is a quantity we often want to measure accurately. In order to know

at what time of day a particular event occurred at a particular computer it is necessary to

synchronize its clock with an authoritative, external source of time. For example, an

eCommerce transaction involves events at a merchant’s computer and at a bank’s

computer. It is important, for auditing purposes, that those events are timestamped

accurately.

Second, algorithms that depend upon clock synchronization have been developed

for several problems in distribution [Liskov 1993]. These include maintaining the

consistency of distributed data (the use of timestamps to serialize transactions is

discussed in Section 16.6), checking the authenticity of a request sent to a server (a

version of the Kerberos authentication protocol, discussed in Chapter 11, depends on

loosely synchronized clocks) and eliminating the processing of duplicate updates (see,

for example, Ladin et al. [1992]).

Measuring time can be problematic due to the existence of multiple frames of

reference. Einstein demonstrated, in his Special Theory of Relativity, the intriguing

consequences that follow from the observation that the speed of light is constant for all

observers, regardless of their relative velocity. He proved from this assumption, among

other things, that two events that are judged to be simultaneous in one frame of reference

are not necessarily simultaneous according to observers in other frames of reference that

are moving relative to it. For example, an observer on the Earth and an observer

travelling away from the Earth in a spaceship will disagree on the time interval between

events, the more so as their relative speed increases.

The relative order of two events can even be reversed for two different observers.

But this cannot happen if one event causes the other to occur. In that case, the physical

effect follows the physical cause for all observers, although the time elapsed between

cause and effect can vary. The timing of physical events was thus proved to be relative

to the observer, and Newton’s notion of absolute physical time was shown to be without

foundation. There is no special physical clock in the universe to which we can appeal

when we want to measure intervals of time.

The notion of physical time is also problematic in a distributed system. This is not

due to the effects of special relativity, which are negligible or nonexistent for normal

computers (unless one counts computers travelling in spaceships!). Rather, the problem

is based on a similar limitation in our ability to timestamp events at different nodes

sufficiently accurately to know the order in which any pair of events occurred, or

whether they occurred simultaneously. There is no absolute, global time to which we

can appeal. And yet we sometimes need to observe distributed systems and establish

whether certain states of affairs occurred at the same time. For example, in objectoriented systems we need to be able to establish whether references to a particular object

no longer exist – whether the object has become garbage (in which case we can free its

memory). Establishing this requires observations of the states of processes (to find outSECTION 14.2 CLOCKS, EVENTS AND PROCESS STATES 613

whether they contain references) and of the communication channels between processes

(in case messages containing references are in transit).

In the first half of this chapter, we examine methods whereby computer clocks can

be approximately synchronized, using message passing. We go on to introduce logical

clocks, including vector clocks, which are used to define an order of events without

measuring the physical time at which they occurred.

In the second half, we describe algorithms whose purpose is to capture global

states of distributed systems as they execute.

14.2 Clocks, events and process states

Chapter 2 presented an introductory model of interaction between the processes within

a distributed system. Here we refine that model in order to help us to understand how to

characterize the system’s evolution as it executes, and how to timestamp the events in a

system’s execution that interest users. We begin by considering how to order and

timestamp the events that occur at a single process.

We take a distributed system to consist of a collection ℘ of N processes

Each process executes on a single processor, and the processors do

not share memory (Chapter 6 briefly considered the case of processes that share

memory). Each process in ℘ has a state that, in general, it transforms as it

executes. The process’s state includes the values of all the variables within it. Its state

may also include the values of any objects in its local operating system environment that

it affects, such as files. We assume that processes cannot communicate with one another

in any way except by sending messages through the network. So, for example, if the

processes operate robot arms connected to their respective nodes in the system, then they

are not allowed to communicate by shaking one another’s robot hands!

As each process executes it takes a series of actions, each of which is either a

message send or receive operation, or an operation that transforms ’s state – one that

changes one or more of the values in In practice, we may choose to use a high-level

description of the actions, according to the application. For example, if the processes in

℘ are engaged in an eCommerce application, then the actions may be ones such as

‘client dispatched order message’ or ‘merchant server recorded transaction to log’.

We define an event to be the occurrence of a single action that a process carries

out as it executes – a communication action or a state-transforming action. The sequence

of events within a single process can be placed in a single, total ordering, which we

denote by the relation →i between the events. That is, e →i e' if and only if the event

occurs before at . This ordering is well defined, whether or not the process is multithreaded, since we have assumed that the process executes on a single processor.

Now we can define the history of process to be the series of events that take

place within it, ordered as we have described by the relation →i:

Clocks • We have seen how to order the events at a process, but not how to timestamp

them – i.e., to assign to them a date and time of day. Computers each contain their own

physical clocks. These clocks are electronic devices that count oscillations occurring in

pi, i = 1 2 , , …N.

pi si

pi

pi

s

i.

pi

e

e′ pi

pi

history(pi ) = = hi <ei 0, , , ei 1 ei 2 …>614 CHAPTER 14 TIME AND GLOBAL STATES

a crystal at a definite frequency, and typically divide this count and store the result in a

counter register. Clock devices can be programmed to generate interrupts at regular

intervals in order that, for example, timeslicing can be implemented; however, we shall

not concern ourselves with this aspect of clock operation.

The operating system reads the node’s hardware clock value, , scales it and

adds an offset so as to produce a software clock that approximately

measures real, physical time t for process . In other words, when the real time in an

absolute frame of reference is t, is the reading on the software clock. For example,

could be the 64-bit value of the number of nanoseconds that have elapsed at time

t since a convenient reference time. In general, the clock is not completely accurate, so

will differ from t. Nonetheless, if behaves sufficiently well (we shall examine

the notion of clock correctness shortly), we can use its value to timestamp any event at

. Note that successive events will correspond to different timestamps only if the clock

resolution – the period between updates of the clock value – is smaller than the time

interval between successive events. The rate at which events occur depends on such

factors as the length of the processor instruction cycle.

Clock skew and clock drift • Computer clocks, like any others, tend not to be in perfect

agreement (Figure 14.1). The instantaneous difference between the readings of any two

clocks is called their skew. Also, the crystal-based clocks used in computers are, like any

other clocks, subject to clock drift, which means that they count time at different rates,

and so diverge. The underlying oscillators are subject to physical variations, with the

consequence that their frequencies of oscillation differ. Moreover, even the same

clock’s frequency varies with temperature. Designs exist that attempt to compensate for

this variation, but they cannot eliminate it. The difference in the oscillation period

between two clocks might be extremely small, but the difference accumulated over

many oscillations leads to an observable difference in the counters registered by two

clocks, no matter how accurately they were initialized to the same value. A clock’s drift

rate is the change in the offset (difference in reading) between the clock and a nominal

perfect reference clock per unit of time measured by the reference clock. For ordinary

clocks based on a quartz crystal this is about 10–6 seconds/second, giving a difference

of 1 second every 1,000,000 seconds, or 11.6 days. The drift rate of ‘high-precision’

quartz clocks is about 10–7 or 10–8.

Coordinated Universal Time • Computer clocks can be synchronized to external sources

of highly accurate time. The most accurate physical clocks use atomic oscillators, whose

drift rate is about one part in 1013. The output of these atomic clocks is used as the

Hi

( ) t

C

i( ) α t = Hi( ) β t +

pi

C

i( ) t

C

i( ) t

C

i( ) t Ci

pi

Figure 14.1 Skew between computer clocks in a distributed system

NetworkSECTION 14.3 SYNCHRONIZING PHYSICAL CLOCKS 615

standard for elapsed real time, known as International Atomic Time. Since 1967, the

standard second has been defined as 9,192,631,770 periods of transition between the

two hyperfine levels of the ground state of Caesium-133 (Cs133).

Seconds and years and other time units that we use are rooted in astronomical

time. They were originally defined in terms of the rotation of the Earth on its axis and

its rotation about the Sun. However, the period of the Earth’s rotation about its axis is

gradually getting longer, primarily because of tidal friction; atmospheric effects and

convection currents within the Earth’s core also cause short-term increases and

decreases in the period. So astronomical time and atomic time have a tendency to get out

of step.

Coordinated Universal Time – abbreviated as UTC (from the French equivalent)

– is an international standard for timekeeping. It is based on atomic time, but a so-called

‘leap second’ is inserted – or, more rarely, deleted – occasionally to keep it in step with

astronomical time. UTC signals are synchronized and broadcast regularly from landbased radio stations and satellites covering many parts of the world. For example, in the

USA, the radio station WWV broadcasts time signals on several shortwave frequencies.

Satellite sources include the Global Positioning System (GPS).

Receivers are available commercially. Compared with ‘perfect’ UTC, the signals

received from land-based stations have an accuracy on the order of 0.1–10 milliseconds,

depending on the station used. Signals received from GPS satellites are accurate to about

1 microsecond. Computers with receivers attached can synchronize their clocks with

these timing signals.

14.3 Synchronizing physical clocks

In order to know at what time of day events occur at the processes in our distributed

system ℘ – for example, for accountancy purposes – it is necessary to synchronize the

processes’ clocks, , with an authoritative, external source of time. This is external

synchronization. And if the clocks are synchronized with one another to a known

degree of accuracy, then we can measure the interval between two events occurring at

different computers by appealing to their local clocks, even though they are not

necessarily synchronized to an external source of time. This is internal synchronization.

We define these two modes of synchronization more closely as follows, over an interval

of real time I:

External synchronization: For a synchronization bound , and for a source S of

UTC time, < D, for and for all real times t in I. Another

way of saying this is that the clocks are accurate to within the bound D.

Internal synchronization: For a synchronization bound ,

for , and for all real times t in I. Another way of saying this is that

the clocks agree within the bound D.

Clocks that are internally synchronized are not necessarily externally synchronized,

since they may drift collectively from an external source of time even though they agree

with one another. However, it follows from the definitions that if the system ℘ is

C

i

C

i

D > 0

S t ( ) – Ci( ) t i = 1 2 , , …N

C

i

D > 0 C

i( ) t – Cj( ) t < D

i j , = 1 2 , , …N

C

i616 CHAPTER 14 TIME AND GLOBAL STATES

externally synchronized with a bound D, then the same system is internally

synchronized with a bound of 2D.

Various notions of correctness for clocks have been suggested. It is common to

define a hardware clock H to be correct if its drift rate falls within a known bound ρ > 0

(a value derived from one supplied by the manufacturer, such as 10–6 seconds/second).

This means that the error in measuring the interval between real times t and ( ) is

bounded:

This condition forbids jumps in the value of hardware clocks (during normal operation).

Sometimes we also require our software clocks to obey the condition but a weaker

condition of monotonicity may suffice. Monotonicity is the condition that a clock C only

ever advances:

For example, the UNIX make facility is a tool that is used to compile only those source

files that have been modified since they were last compiled. The modification dates of

each corresponding pair of source and object files are compared to determine this

condition. If a computer whose clock was running fast set its clock back after compiling

a source file but before the file was changed, the source file might appear to have been

modified prior to the compilation. Erroneously, make will not recompile the source file.

We can achieve monotonicity despite the fact that a clock is found to be running

fast. We need only change the rate at which updates are made to the time as given to

applications. This can be achieved in software without changing the rate at which the

underlying hardware clock ticks – recall that , where we are free to

choose the values of and .

A hybrid correctness condition that is sometimes applied is to require that a clock

obeys the monotonicity condition, and that its drift rate is bounded between

synchronization points, but to allow the clock value to jump ahead at synchronization

points.

A clock that does not keep to whatever correctness conditions apply is defined to

be faulty. A clock’s crash failure is said to occur when the clock stops ticking altogether;

any other clock failure is an arbitrary failure. A historical example of an arbitrary failure

is that of a clock with the ‘Y2K bug’, which broke the monotonicity condition by

registering the date after 31 December 1999 as 1 January 1900 instead of 2000; another

example is a clock whose batteries are very low and whose drift rate suddenly becomes

very large.

Note that clocks do not have to be accurate to be correct, according to the

definitions. Since the goal may be internal rather than external synchronization, the

criteria for correctness are only concerned with the proper functioning of the clock’s

‘mechanism’, not its absolute setting.

We now describe algorithms for external synchronization and for internal

synchronization.

t′ t′ > t

( ) 1 – ρ ( ) t′ – t ≤ H t ( ) ′ – H t ( ) ≤ ( ) 1 + ρ ( ) t′ – t

t′ > t  C t ( ) ′ > C t ( )

C

i( ) α t = Hi( ) β t +

α βSECTION 14.3 SYNCHRONIZING PHYSICAL CLOCKS 617

14.3.1 Synchronization in a synchronous system

We begin by considering the simplest possible case: of internal synchronization between

two processes in a synchronous distributed system. In a synchronous system, bounds are

known for the drift rate of clocks, the maximum message transmission delay, and the

time required to execute each step of a process (see Section 2.4.1).

One process sends the time t on its local clock to the other in a message m. In

principle, the receiving process could set its clock to the time , where is

the time taken to transmit m between them. The two clocks would then agree (since the

aim is internal synchronization, it does not matter whether the sending process’s clock

is accurate).

Unfortunately, is subject to variation and is unknown. In general, other

processes are competing for resources with the processes to be synchronized at their

respective nodes, and other messages compete with m for the network resources.

Nonetheless, there is always a minimum transmission time, min, that would be obtained

if no other processes executed and no other network traffic existed; min can be measured

or conservatively estimated.

In a synchronous system, by definition, there is also an upper bound max on the

time taken to transmit any message. Let the uncertainty in the message transmission time

be u, so that . If the receiver sets its clock to be , then the

clock skew may be as much as u, since the message may in fact have taken time max to

arrive. Similarly, if it sets its clock to , the skew may again be as large as u. If,

however, it sets its clock to the halfway point, , then the skew is at

most . In general, for a synchronous system, the optimum bound that can be

achieved on clock skew when synchronizing N clocks is [Lundelius and

Lynch 1984].

Most distributed systems found in practice are asynchronous: the factors leading

to message delays are not bounded in their effect, and there is no upper bound max on

message transmission delays. This is particularly so for the Internet. For an

asynchronous system, we may say only that , where x  0. The value

of x is not known in a particular case, although a distribution of values may be

measurable for a particular installation.

14.3.2 Cristian’s method for synchronizing clocks

Cristian [1989] suggested the use of a time server, connected to a device that receives

signals from a source of UTC, to synchronize computers externally. Upon request, the

server process S supplies the time according to its clock, as shown in Figure 14.2.

t T

+ trans Ttrans

T

trans

u max min = ( ) – t min +

t max +

t max min + ( ) + ⁄ 2

u ⁄ 2

u( ) 1 1 – ⁄ N

T

trans = min x +

Figure 14.2 Clock synchronization using a time server

m

r

m t

P Time Server, S

mr

mt

p Time server, S618 CHAPTER 14 TIME AND GLOBAL STATES

Cristian observed that while there is no upper bound on message transmission delays in

an asynchronous system, the round-trip times for messages exchanged between pairs of

processes are often reasonably short – a small fraction of a second. He describes the

algorithm as probabilistic: the method achieves synchronization only if the observed

round-trip times between client and server are sufficiently short compared with the

required accuracy.

A process p requests the time in a message , and receives the time value t in a

message (t is inserted in at the last possible point before transmission from S’s

computer). Process p records the total round-trip time taken to send the request

and receive the reply . It can measure this time with reasonable accuracy if its

rate of clock drift is small. For example, the round-trip time should be on the order of 1–

10 milliseconds on a LAN, over which time a clock with a drift rate of 10–6

seconds/second varies by at most 10–5 milliseconds.

A simple estimate of the time to which p should set its clock is ,

which assumes that the elapsed time is split equally before and after S placed t in .

This is normally a reasonably accurate assumption, unless the two messages are

transmitted over different networks. If the value of the minimum transmission time min

is known or can be conservatively estimated, then we can determine the accuracy of this

result as follows.

The earliest point at which S could have placed the time in was min after p

dispatched . The latest point at which it could have done this was min before

arrived at p. The time by S’s clock when the reply message arrives is therefore in the

range . The width of this range is , so the

accuracy is .

Variability can be dealt with to some extent by making several requests to S

(spacing the requests so that transitory congestion can clear) and taking the minimum

value of to give the most accurate estimate. The greater the accuracy required,

the smaller the probability of achieving it. This is because the most accurate results are

those in which both messages are transmitted in a time close to min – an unlikely event

in a busy network.

Discussion of Cristian’s algorithm • As described, Cristian’s method suffers from the

problem associated with all services implemented by a single server: that the single time

server might fail and thus render synchronization temporarily impossible. Cristian

suggested, for this reason, that time should be provided by a group of synchronized time

servers, each with a receiver for UTC time signals. For example, a client could multicast

its request to all servers and use only the first reply obtained.

Note that a faulty time server that replied with spurious time values, or an imposter

time server that replied with deliberately incorrect times, could wreak havoc in a

computer system. These problems were beyond the scope of the work described by

Cristian [1989], which assumes that sources of external time signals are self-checking.

Cristian and Fetzer [1994] describe a family of probabilistic protocols for internal clock

synchronization, each of which tolerates certain failures. Srikanth and Toueg [1987]

first described an algorithm that is optimal with respect to the accuracy of the

synchronized clocks, while tolerating some failures. Dolev et al. [1986] showed that if

f is the number of faulty clocks out of a total of N, then we must have if the other,

correct, clocks are still to be able to achieve agreement. The problem of dealing with

m

r

m

t mt

T

round

m

r mt

t T

+ round ⁄ 2

m

t

m

t

m

r mt

t min + t T

[ , ] + round – min Tround – 2min

± T

( ) round ⁄ 2 – min

T

round

N > 3fSECTION 14.3 SYNCHRONIZING PHYSICAL CLOCKS 619

faulty clocks is partially addressed by the Berkeley algorithm, which is described next.

The problem of malicious interference with time synchronization can be dealt with by

authentication techniques.

14.3.3 The Berkeley algorithm

Gusella and Zatti [1989] describe an algorithm for internal synchronization that they

developed for collections of computers running Berkeley UNIX. In it, a coordinator

computer is chosen to act as the master. Unlike in Cristian’s protocol, this computer

periodically polls the other computers whose clocks are to be synchronized, called

slaves. The slaves send back their clock values to it. The master estimates their local

clock times by observing the round-trip times (similarly to Cristian’s technique), and it

averages the values obtained (including its own clock’s reading). The balance of

probabilities is that this average cancels out the individual clocks’ tendencies to run fast

or slow. The accuracy of the protocol depends upon a nominal maximum round-trip time

between the master and the slaves. The master eliminates any occasional readings

associated with larger times than this maximum.

Instead of sending the updated current time back to the other computers – which

would introduce further uncertainty due to the message transmission time – the master

sends the amount by which each individual slave’s clock requires adjustment. This can

be a positive or negative value.

The Berkeley algorithm eliminates readings from faulty clocks. Such clocks could

have a significant adverse effect if an ordinary average was taken so instead the master

takes a fault-tolerant average. That is, a subset is chosen of clocks that do not differ from

one another by more than a specified amount, and the average is taken of readings from

only these clocks.

Gusella and Zatti describe an experiment involving 15 computers whose clocks

were synchronized to within about 20–25 milliseconds using their protocol. The local

clocks’ drift rates were measured to be less than 2×10–5, and the maximum round-trip

time was taken to be 10 milliseconds.

Should the master fail, then another can be elected to take over and function

exactly as its predecessor. Section 15.3 discusses some general-purpose election

algorithms. Note that these are not guaranteed to elect a new master in bounded time, so

the difference between two clocks would be unbounded if they were used.

14.3.4 The Network Time Protocol

Cristian’s method and the Berkeley algorithm are intended primarily for use within

intranets. The Network Time Protocol (NTP) [Mills 1995] defines an architecture for a

time service and a protocol to distribute time information over the Internet.

NTP’s chief design aims and features are as follows:

To provide a service enabling clients across the Internet to be synchronized

accurately to UTC: Although large and variable message delays are encountered in

Internet communication, NTP employs statistical techniques for the filtering of

timing data and it discriminates between the quality of timing data from different

servers.620 CHAPTER 14 TIME AND GLOBAL STATES

To provide a reliable service that can survive lengthy losses of connectivity: There

are redundant servers and redundant paths between the servers. The servers can

reconfigure so as to continue to provide the service if one of them becomes

unreachable.

To enable clients to resynchronize sufficiently frequently to offset the rates of drift

found in most computers: The service is designed to scale to large numbers of clients

and servers.

To provide protection against interference with the time service, whether malicious

or accidental: The time service uses authentication techniques to check that timing

data originate from the claimed trusted sources. It also validates the return addresses

of messages sent to it.

The NTP service is provided by a network of servers located across the Internet. Primary

servers are connected directly to a time source such as a radio clock receiving UTC;

secondary servers are synchronized, ultimately, with primary servers. The servers are

connected in a logical hierarchy called a synchronization subnet (see Figure 14.3),

whose levels are called strata. Primary servers occupy stratum 1: they are at the root.

Stratum 2 servers are secondary servers that are synchronized directly with the primary

servers; stratum 3 servers are synchronized with stratum 2 servers, and so on. The

lowest-level (leaf) servers execute in users’ workstations.

The clocks belonging to servers with high stratum numbers are liable to be less

accurate than those with low stratum numbers, because errors are introduced at each

level of synchronization. NTP also takes into account the total message round-trip

delays to the root in assessing the quality of timekeeping data held by a particular server.

The synchronization subnet can reconfigure as servers become unreachable or

failures occur. If, for example, a primary server’s UTC source fails, then it can become

Figure 14.3 An example synchronization subnet in an NTP implementation

1

2

3

2

3 3

Arrows denote synchronization control, numbers denote strata.SECTION 14.3 SYNCHRONIZING PHYSICAL CLOCKS 621

a stratum 2 secondary server. If a secondary server’s normal source of synchronization

fails or becomes unreachable, then it may synchronize with another server.

NTP servers synchronize with one another in one of three modes: multicast,

procedure-call and symmetric mode. Multicast mode is intended for use on a high-speed

LAN. One or more servers periodically multicasts the time to the servers running in

other computers connected by the LAN, which set their clocks assuming a small delay.

This mode can achieve only relatively low accuracies, but ones that nonetheless are

considered sufficient for many purposes.

Procedure-call mode is similar to the operation of Cristian’s algorithm, described

in Section 14.3.2. In this mode, one server accepts requests from other computers, which

it processes by replying with its timestamp (current clock reading). This mode is suitable

where higher accuracies are required than can be achieved with multicast, or where

multicast is not supported in hardware. For example, file servers on the same or a

neighbouring LAN that need to keep accurate timing information for file accesses could

contact a local server in procedure-call mode.

Finally, symmetric mode is intended for use by the servers that supply time

information in LANs and by the higher levels (lower strata) of the synchronization

subnet, where the highest accuracies are to be achieved. A pair of servers operating in

symmetric mode exchange messages bearing timing information. Timing data are

retained as part of an association between the servers that is maintained in order to

improve the accuracy of their synchronization over time.

In all modes, messages are delivered unreliably, using the standard UDP Internet

transport protocol. In procedure-call mode and symmetric mode, processes exchange

pairs of messages. Each message bears timestamps of recent message events: the local

times when the previous NTP message between the pair was sent and received, and the

local time when the current message was transmitted. The recipient of the NTP message

notes the local time when it receives the message. The four times , ,

and are shown in Figure 14.4 for the messages m and m' sent between servers A and

B. Note that in symmetric mode, unlike in Cristian’s algorithm, there can be a nonnegligible delay between the arrival of one message and the dispatch of the next. Also,

messages may be lost, but the three timestamps carried by each message are nonetheless

valid.

Ti

– 3 Ti – 2 Ti – 1

Ti

Figure 14.4 Messages exchanged between a pair of NTP peers

Ti

Ti–2 Ti–1

Ti–3

Server B

Server A

Time

m m'

Time622 CHAPTER 14 TIME AND GLOBAL STATES

For each pair of messages sent between two servers the NTP calculates an offset

, which is an estimate of the actual offset between the two clocks, and a delay ,

which is the total transmission time for the two messages. If the true offset of the clock

at B relative to that at A is o, and if the actual transmission times for m and m' are t and

t', respectively, then we have:

and

This leads to:

and:

, where

Using the fact that , it can be shown that . Thus is

an estimate of the offset, and is a measure of the accuracy of this estimate.

NTP servers apply a data filtering algorithm to successive pairs which

estimates the offset o and calculates the quality of this estimate as a statistical quantity

called the filter dispersion. A relatively high filter dispersion represents relatively

unreliable data. The eight most recent pairs are retained. As with Cristian’s

algorithm, the value of that corresponds to the minimum value is chosen to estimate

o.

The value of the offset derived from communication with a single source is not

necessarily used by itself to control the local clock, however. In general, an NTP server

engages in message exchanges with several of its peers. In addition to data filtering

applied to exchanges with each single peer, NTP applies a peer-selection algorithm. This

examines the values obtained from exchanges with each of several peers, looking for

relatively unreliable values. The output from this algorithm may cause a server to

change the peer that it primarily uses for synchronization.

Peers with lower stratum numbers are more favoured than those in higher strata

because they are ‘closer’ to the primary time sources. Also, those with the lowest

synchronization dispersion are relatively favoured. This is the sum of the filter

dispersions measured between the server and the root of the synchronization subnet.

(Peers exchange synchronization dispersions in messages, allowing this total to be

calculated.)

NTP employs a phase lock loop model [Mills 1995], which modifies the local

clock’s update frequency in accordance with observations of its drift rate. To take a

simple example, if a clock is discovered always to gain time at the rate of, say, four

seconds per hour, then its frequency can be reduced slightly (in software or hardware)

to compensate for this. The clock’s drift in the intervals between synchronization is thus

reduced.

Mills quotes synchronization accuracies on the order of tens of milliseconds over

Internet paths, and one millisecond on LANs.

o

i di

Ti

– 2 = Ti – 3 + + t o Ti = Ti – 1 + t′ – o

d

i = = t t + ′ Ti – 2 – Ti – 3 + Ti – Ti – 1

o o

= i + ( ) t′ – t ⁄ 2 oi = ( ) Ti – 2 – Ti – 3 + Ti – 1 – Ti ⁄ 2

t t , ′ ≥ 0 oi – di ⁄ 2 ≤ ≤ o oi + di ⁄ 2 oi

d

i

<o

i, di>,

<o

i, di>

o

j djSECTION 14.4 LOGICAL TIME AND LOGICAL CLOCKS 623

14.4 Logical time and logical clocks

From the point of view of any single process, events are ordered uniquely by times

shown on the local clock. However, as Lamport [1978] pointed out, since we cannot

synchronize clocks perfectly across a distributed system, we cannot in general use

physical time to find out the order of any arbitrary pair of events occurring within it.

In general, we can use a scheme that is similar to physical causality but that applies

in distributed systems to order some of the events that occur at different processes. This

ordering is based on two simple and intuitively obvious points:

• If two events occurred at the same process , then they

occurred in the order in which observes them – this is the order →i that we

defined above.

• Whenever a message is sent between processes, the event of sending the message

occurred before the event of receiving the message.

Lamport called the partial ordering obtained by generalizing these two relationships the

happened-before relation. It is also sometimes known as the relation of causal ordering

or potential causal ordering.

We can define the happened-before relation, denoted by →, as follows:

HB1: If ∃ process : e →i e', then .

HB2: For any message m, send(m) → receive(m)

– where send(m) is the event of sending the message, and receive(m)

is the event of receiving it.

HB3: If e, and are events such that and , then .

Thus, if e and are events, and if , then we can find a series of events

occurring at one or more processes such that and , and

for either HB1 or HB2 applies between and . That is, either

they occur in succession at the same process, or there is a message m such that =

send(m) and = receive(m). The sequence of events need not be

unique.

The relation → is illustrated for the case of three processes, , and , in

Figure 14.5. It can be seen that a → b, since the events occur in this order at process

(a →i b), and similarly c → d. Furthermore, b → c, since these events are the sending and

pi ( ) i = 1 2 , , , … N

pi

pi e e → ′

e′ e″ e e → ′ e′ → e″ e e → ″

e′ e e → ′

e

1, , , e2 … en e e = 1 e′ = en

i = 1 2 , , …, N – 1 ei ei ·+ 1

e

i

e

i + 1 e1, , , e2 … en

p1 p2 p3

Figure 14.5 Events occurring at three processes

p1

p2

p3

a b

c d

e f

m1

m2

Physical

time

p1624 CHAPTER 14 TIME AND GLOBAL STATES

reception of message , and similarly d → f. Combining these relations, we may also

say that, for example, a → f.

It can also be seen from Figure 14.5 that not all events are related by the relation

→. For example, and , since they occur at different processes, and there is

no chain of messages intervening between them. We say that events such as a and e that

are not ordered by → are concurrent and write this .

The relation → captures a flow of data intervening between two events. Note,

however, that in principle data can flow in ways other than by message passing. For

example, if Smith enters a command to his process to send a message, then telephones

Jones, who commands her process to issue another message, the issuing of the first

message clearly happened-before that of the second. Unfortunately, since no network

messages were sent between the issuing processes, we cannot model this type of

relationship in our system.

Another point to note is that if the happened-before relation holds between two

events, then the first might or might not actually have caused the second. For example,

if a server receives a request message and subsequently sends a reply, then clearly the

reply transmission is caused by the request transmission. However, the relation →

captures only potential causality, and two events can be related by → even though there

is no real connection between them. A process might, for example, receive a message

and subsequently issue another message, but one that it issues every five minutes

anyway and that bears no specific relation to the first message. No actual causality has

been involved, but the relation → would order these events.

Logical clocks • Lamport [1978] invented a simple mechanism by which the happenedbefore ordering can be captured numerically, called a logical clock. A Lamport logical

clock is a monotonically increasing software counter, whose value need bear no

particular relationship to any physical clock. Each process keeps its own logical

clock, , which it uses to apply so-called Lamport timestamps to events. We denote the

timestamp of event e at by , and by we denote the timestamp of event e

at whatever process it occurred at.

To capture the happened-before relation →, processes update their logical clocks

and transmit the values of their logical clocks in messages as follows:

LC1: is incremented before each event is issued at process :

:=

LC2: (a) When a process sends a message m, it piggybacks on m the value

.

(b) On receiving (m, t), a process computes := and then

applies LC1 before timestamping the event receive(m).

Although we increment clocks by 1, we could have chosen any positive value. It can

easily be shown, by induction on the length of any sequence of events relating two

events e and , that .

Note that the converse is not true. If , then we cannot infer that

. In Figure 14.6 we illustrate the use of logical clocks for the example given in

Figure 14.5. Each of the processes , and has its logical clock initialized to 0.

The clock values given are those immediately after the event to which they are adjacent.

Note that, for example, but .

m

1

a e →/ e a →/

a e ||

pi

L

i

pi Li( ) e L e ( )

L

i pi

L

i Li + 1.

pi

t L

= i

pj Lj max L ( ) j, t

e′ e e → ′  L e ( ) < L e ( ) ′

L e ( ) < L e ( ) ′

e e → ′

p1 p2 p3

L b ( ) > L e ( ) b e ||SECTION 14.4 LOGICAL TIME AND LOGICAL CLOCKS 625

Totally ordered logical clocks • Some pairs of distinct events, generated by different

processes, have numerically identical Lamport timestamps. However, we can create a

total order on the set of events – that is, one for which all pairs of distinct events are

ordered – by taking into account the identifiers of the processes at which events occur.

If e is an event occurring at with local timestamp , and is an event occurring at

with local timestamp , we define the global logical timestamps for these events to

be and , respectively. And we define if and only if either

, or and . This ordering has no general physical significance

(because process identifiers are arbitrary), but it is sometimes useful. Lamport used it,

for example, to order the entry of processes to a critical section.

Vector clocks • Mattern [1989] and Fidge [1991] developed vector clocks to overcome

the shortcoming of Lamport’s clocks: the fact that from we cannot

conclude that . A vector clock for a system of N processes is an array of N

integers. Each process keeps its own vector clock, , which it uses to timestamp local

events. Like Lamport timestamps, processes piggyback vector timestamps on the

messages they send to one another, and there are simple rules for updating the clocks:

VC1: Initially, , for .

VC2: Just before timestamps an event, it sets :=

VC3: includes the value in every message it sends.

VC4: When receives a timestamp t in a message, it sets

:= , for . Taking the componentwise maximum of two vector timestamps in this way is known as a merge

operation.

For a vector clock , is the number of events that has timestamped, and

is the number of events that have occurred at that have potentially

affected . (Process may have timestamped more events by this point, but no

information has flowed to about them in messages as yet.)

Figure 14.6 Lamport timestamps for the events shown in Figure 14.5

a b

c d

e f

m1

m2

1 2

3 4

1 5

p1

p2

p3

Physical

time

pi Ti e′

pj Tj

Ti

( ) , i ( ) Tj, j ( ) Ti, i < ( ) Tj, j

Ti

Tj

< T

i = Tj i j <

L e ( ) < L e ( ) ′

e e′

·

→

Vi

Vi

[ ] j = 0 i j , = 1 2 , , … N

pi Vi[ ] i Vi[ ] i + 1.

pi t V = i

pi

Vi

[ ] j max V ( ) i[ ] j , t j [ ] j = 1 2 , , … N

Vi

Vi

[ ] i pi

Vi

[ ] j j i ( ) ≠ pj

pi pj

pi626 CHAPTER 14 TIME AND GLOBAL STATES

We may compare vector timestamps as follows:

for

for

Let be the vector timestamp applied by the process at which e occurs. It is

straightforward to show, by induction on the length of any sequence of events relating

two events e and , that . Exercise 10.13 leads the reader to

show the converse: if , then .

Figure 14.7 shows the vector timestamps of the events of Figure 14.5. It can be

seen, for example, that , which reflects the fact that a → f. Similarly, we

can tell when two events are concurrent by comparing their timestamps. For example,

that can be seen from the facts that neither nor .

Vector timestamps have the disadvantage, compared with Lamport timestamps, of

taking up an amount of storage and message payload that is proportional to N, the

number of processes. Charron-Bost [1991] showed that, if we are to be able to tell

whether or not two events are concurrent by inspecting their timestamps, then the

dimension N is unavoidable. However, techniques exist for storing and transmitting

smaller amounts of data, at the expense of the processing required to reconstruct

complete vectors. Raynal and Singhal [1996] give an account of some of these

techniques. They also describe the notion of matrix clocks, whereby processes keep

estimates of other processes’ vector times as well as their own.

14.5 Global states

In this and the next section we examine the problem of finding out whether a particular

property is true of a distributed system as it executes. We begin by giving the examples

of distributed garbage collection, deadlock detection, termination detection and

debugging:

V V = = ′ iff V j [ ] V′[ ] j j = 1 2 , , … N

V V ≤ ′ iff V j [ ] ≤ V′[ ] j j = 1 2 , , … N

V V < ′ iff V V ≤ ′ ∧ V V ≠ ′

Figure 14.7 Vector timestamps for the events shown in Figure 14.5

a b

c d

e f

m1

m2

(1,0,0) (2,0,0)

(2,1,0) (2,2,0)

(0,0,1) (2,2,2)

p1

p2

p3

Physical

time

V e ( )

e′ e e → ′  V e ( ) < V e ( ) ′

V e ( ) < V e ( ) ′ e e → ′

V a ( ) < V f ( )

c e || V c ( ) ≤ V e ( ) V e ( ) ≤ V c ( )SECTION 14.5 GLOBAL STATES 627

Distributed garbage collection: An object is considered to be garbage if there are no

longer any references to it anywhere in the distributed system. The memory taken up

by that object can be reclaimed once it is known to be garbage. To check that an

object is garbage, we must verify that there are no references to it anywhere in the

system. In Figure 14.8(a), process has two objects that both have references – one

has a reference within itself, and has a reference to the other. Process has

one garbage object, with no references to it anywhere in the system. It also has an

object for which neither nor has a reference, but there is a reference to it in a

message that is in transit between the processes. This shows that when we consider

properties of a system, we must include the state of communication channels as well

as the state of the processes.

Distributed deadlock detection: A distributed deadlock occurs when each of a

collection of processes waits for another process to send it a message, and where

there is a cycle in the graph of this ‘waits-for’ relationship. Figure 14.8(b) shows that

processes and are each waiting for a message from the other, so this system

will never make progress.

Distributed termination detection: The problem here is how to detect that a

distributed algorithm has terminated. Detecting termination is a problem that sounds

deceptively easy to solve: it seems at first only necessary to test whether each process

has halted. To see that this is not so, consider a distributed algorithm executed by two

processes and , each of which may request values from the other.

Instantaneously, we may find that a process is either active or passive – a passive

process is not engaged in any activity of its own but is prepared to respond with a

value requested by the other. Suppose we discover that is passive and that is

p1

p1 p2 p2

p1 p2

Figure 14.8 Detecting global properties

p1 p2

message

garbage object

object

reference

(a) Garbage collection

p1 wait-for p2

(b) Deadlock wait-for

p1 p2

activate

(c) Termination passive passive

p1 p2

p1 p2

p1 p2628 CHAPTER 14 TIME AND GLOBAL STATES

passive (Figure 14.8c). To see that we may not conclude that the algorithm has

terminated, consider the following scenario: when we tested for passivity, a

message was on its way from , which became passive immediately after sending

it. On receipt of the message, became active again – after we had found it to be

passive. The algorithm had not terminated.

The phenomena of termination and deadlock are similar in some ways, but they

are different problems. First, a deadlock may affect only a subset of the processes in

a system, whereas all processes must have terminated. Second, process passivity is

not the same as waiting in a deadlock cycle: a deadlocked process is attempting to

perform a further action, for which another process waits; a passive process is not

engaged in any activity.

Distributed debugging: Distributed systems are complex to debug [Bonnaire et al.

1995], and care needs to be taken in establishing what occurred during the execution.

For example, suppose Smith has written an application in which each process

contains a variable ( ). The variables change as the program

executes, but they are required always to be within a value δ of one another.

Unfortunately, there is a bug in the program, and Smith suspects that under certain

circumstances for some i and j, breaking her consistency constraints. Her

problem is that this relationship must be evaluated for values of the variables that

occur at the same time.

Each of the problems above has specific solutions tailored to it; but they all illustrate the

need to observe a global state, and so motivate a general approach.

14.5.1 Global states and consistent cuts

It is possible in principle to observe the succession of states of an individual process, but

the question of how to ascertain a global state of the system – the state of the collection

of processes – is much harder to address.

The essential problem is the absence of global time. If all processes had perfectly

synchronized clocks, then we could agree on a time at which each process would record

its state – the result would be an actual global state of the system. From the collection of

process states we could tell, for example, whether the processes were deadlocked. But

we cannot achieve perfect clock synchronization, so this method is not available to us.

So we might ask whether we can assemble a meaningful global state from local

states recorded at different real times. The answer is a qualified ‘yes’, but in order to see

this we must first introduce some definitions.

Let us return to our general system ℘ of N processes ( ), whose

execution we wish to study. We said above that a series of events occurs at each process,

and that we may characterize the execution of each process by its history:

Similarly, we may consider any finite prefix of the process’s history:

p1

p2

p1

pi

x

i i = 1 2 , , … N

x

i – xj > δ

pi i = 1 2 , , , … N

history(pi ) = = hi <ei 0, , , ei 1 ei 2 …>

h

ki

<e

0i

e

1i

…e

ki

= , , >SECTION 14.5 GLOBAL STATES 629

Each event either is an internal action of the process (for example, the updating of one

of its variables), or is the sending or receipt of a message over the communication

channels that connect the processes.

In principle, we can record what occurred in ℘’s execution. Each process can

record the events that take place there, and the succession of states it passes through. We

denote by the state of process immediately before the kth event occurs, so that

is the initial state of . We noted in the examples above that the state of the

communication channels is sometimes relevant. Rather than introducing a new type of

state, we make the processes record the sending or receipt of all messages as part of their

state. If we find that process has recorded that it sent a message m to process

, then by examining whether has received that message we can infer whether

or not m is part of the state of the channel between and .

We can also form the global history of ℘ as the union of the individual process

histories:

Mathematically, we can take any set of states of the individual processes to form a global

state . But which global states are meaningful – that is, which

process states could have occurred at the same time? A global state corresponds to initial

prefixes of the individual process histories. A cut of the system’s execution is a subset

of its global history that is a union of prefixes of process histories:

The state in the global state S corresponding to the cut C is that of immediately

after the last event processed by in the cut – ( ). The set of events

{ : } is called the frontier of the cut.

Consider the events occurring at processes and shown in Figure 14.9. The

figure shows two cuts, one with frontier < > and another with frontier < >.

The leftmost cut is inconsistent. This is because at it includes the receipt of the

message , but at it does not include the sending of that message. This is showing

an ‘effect’ without a ‘cause’. The actual execution never was in a global state

corresponding to the process states at that frontier, and we can in principle tell this by

examining the → relation between events. By contrast, the rightmost cut is consistent.

s

ki

pi si 0

pi

pi

pj( ) i j ≠ pj

pi pj

H h

= 0 ∪ ∪ ∪ h1 … hN – 1

S s

= ( ) 1, , s2 …sN

C h

c1

1 h

c2

2

… h

cNN

= ∪ ∪ ∪

s

i pi

pi ei

c

i i = 1 2 , , , … N

e

ci

i i = 1 2 , , , … N

p1 p2

Figure 14.9 Cuts

m1 m2

p1

p2 Physical

time

e01

Consistent cut

Inconsistent cut

e11 e21 e31

e02 e12 e22

e

01

e

02

, e1 2, e2 2

p2

m

1 p1630 CHAPTER 14 TIME AND GLOBAL STATES

It includes both the sending and the receipt of message and the sending but not the

receipt of message . That is consistent with the actual execution – after all, the

message took some time to arrive.

A cut C is consistent if, for each event it contains, it also contains all the events

that happened-before that event:

For all events ,

A consistent global state is one that corresponds to a consistent cut. We may

characterize the execution of a distributed system as a series of transitions between

global states of the system:

In each transition, precisely one event occurs at some single process in the system. This

event is either the sending of a message, the receipt of a message or an internal event. If

two events happened simultaneously, we may nonetheless deem them to have occurred

in a definite order – say, ordered according to process identifiers. (Events that occur

simultaneously must be concurrent: neither happened-before the other.) A system

evolves in this way through consistent global states.

A run is a total ordering of all the events in a global history that is consistent with

each local history’s ordering, . A linearization or consistent run is

an ordering of the events in a global history that is consistent with this happened-before

relation → on H. Note that a linearization is also a run.

Not all runs pass through consistent global states, but all linearizations pass only

through consistent global states. We say that a state is reachable from a state S if

there is a linearization that passes through S and then .

Sometimes we may alter the ordering of concurrent events within a linearization,

and derive a run that still passes through only consistent global states. For example, if

two successive events in a linearization are the receipt of messages by two processes,

then we may swap the order of these two events.

14.5.2 Global state predicates, stability, safety and liveness

Detecting a condition such as deadlock or termination amounts to evaluating a global

state predicate. A global state predicate is a function that maps from the set of global

states of processes in the system ℘ to {True, False}. One of the useful characteristics

of the predicates associated with the state of an object being garbage, of the system being

deadlocked or the system being terminated is that they are all stable: once the system

enters a state in which the predicate is True, it remains True in all future states reachable

from that state. By contrast, when we monitor or debug an application we are often

interested in non-stable predicates, such as that in our example of variables whose

difference is supposed to be bounded. Even if the application reaches a state in which

the bound obtains, it need not stay in that state.

We also note here two further notions relevant to global state predicates: safety

and liveness. Suppose there is an undesirable property that is a predicate of the

system’s global state – for example, could be the property of being deadlocked. Let

m

1

m

2

e C ∈ f e →  f C ∈

S

0 → → → S1 S2 …

∅

i ( ) i = 1 2 , , , … N

S′

S′

α

αSECTION 14.5 GLOBAL STATES 631

be the original state of the system. Safety with respect to is the assertion that

evaluates to False for all states S reachable from . Conversely, let be a desirable

property of a system’s global state – for example, the property of reaching termination.

Liveness with respect to is the property that, for any linearization L starting in the state

, evaluates to True for some state reachable from .

14.5.3 The ‘snapshot’ algorithm of Chandy and Lamport

Chandy and Lamport [1985] describe a ‘snapshot’ algorithm for determining global

states of distributed systems, which we now present. The goal of the algorithm is to

record a set of process and channel states (a ‘snapshot’) for a set of processes

( ) such that, even though the combination of recorded states may never

have occurred at the same time, the recorded global state is consistent.

We shall see that the state that the snapshot algorithm records has convenient

properties for evaluating stable global predicates.

The algorithm records state locally at processes; it does not give a method for

gathering the global state at one site. An obvious method for gathering the state is for all

processes to send the state they recorded to a designated collector process, but we shall

not address this issue further here.

The algorithm assumes that:

• Neither channels nor processes fail – communication is reliable so that every

message sent is eventually received intact, exactly once.

• Channels are unidirectional and provide FIFO-ordered message delivery.

• The graph of processes and channels is strongly connected (there is a path between

any two processes).

• Any process may initiate a global snapshot at any time.

• The processes may continue their execution and send and receive normal

messages while the snapshot takes place.

For each process , let the incoming channels be those at over which other processes

send it messages; similarly, the outgoing channels of are those on which it sends

messages to other processes. The essential idea of the algorithm is as follows. Each

process records its state and also, for each incoming channel, a set of messages sent to

it. The process records, for each channel, any messages that arrived after it recorded its

state and before the sender recorded its own state. This arrangement allows us to record

the states of processes at different times but to account for the differentials between

process states in terms of messages transmitted but not yet received. If process has

sent a message m to process , but has not received it, then we account for m as

belonging to the state of the channel between them.

The algorithm proceeds through use of special marker messages, which are

distinct from any other messages the processes send and which the processes may send

and receive while they proceed with their normal execution. The marker has a dual role:

as a prompt for the receiver to save its own state, if it has not already done so; and as a

means of determining which messages to include in the channel state.

S

0 α α

S

0 β

β

S

0 β SL S0

pi

i = 1 2 , , , … N

pi pi

pi

pi

pj pj632 CHAPTER 14 TIME AND GLOBAL STATES

The algorithm is defined through two rules, the marker receiving rule and the

marker sending rule (Figure 14.10). The marker sending rule obligates processes to send

a marker after they have recorded their state, but before they send any other messages.

The marker receiving rule obligates a process that has not recorded its state to do

so. In that case, this is the first marker that it has received. It notes which messages

subsequently arrive on the other incoming channels. When a process that has already

saved its state receives a marker (on another channel), it records the state of that channel

as the set of messages it has received on it since it saved its state.

Any process may begin the algorithm at any time. It acts as though it has received

a marker (over a nonexistent channel) and follows the marker receiving rule. Thus it

records its state and begins to record messages arriving over all its incoming channels.

Several processes may initiate recording concurrently in this way (as long as the markers

they use can be distinguished).

We illustrate the algorithm for a system of two processes, and , connected

by two unidirectional channels, and . The two processes trade in ‘widgets’.

Process sends orders for widgets over to , enclosing payment at the rate of $10

per widget. Some time later, process sends widgets along channel to . The

Figure 14.10 Chandy and Lamport’s ‘snapshot’ algorithm

Marker receiving rule for process

On receipt of a marker message at over channel c:

if ( has not yet recorded its state) it

records its process state now;

records the state of c as the empty set;

turns on recording of messages arriving over other incoming channels;

else

records the state of c as the set of messages it has received over c

since it saved its state.

end if

Marker sending rule for process

After has recorded its state, for each outgoing channel c:

sends one marker message over c

(before it sends any other message over c).

pi

pi

pi

pi

pi

pi

pi

Figure 14.11 Two processes and their initial states

p1 c2 p2

c1

account widgets

$1000 (none)

account widgets

$50 2000

p1 p2

c

1 c2

p1 c2 p2

p2 c1 p1SECTION 14.5 GLOBAL STATES 633

processes have the initial states shown in Figure 14.11. Process has already received

an order for five widgets, which it will shortly dispatch to .

Figure 14.12 shows an execution of the system while the state is recorded. Process

records its state in the actual global state , when the state of is <$1000, 0>.

Following the marker sending rule, process then emits a marker message over its

outgoing channel before it sends the next application-level message: (Order 10,

$100), over channel . The system enters actual global state .

Before receives the marker, it emits an application message (five widgets) over

in response to ’s previous order, yielding a new actual global state .

Now process receives ’s message (five widgets), and receives the

marker. Following the marker receiving rule, records its state as <$50, 1995> and

that of channel as the empty sequence. Following the marker sending rule, it sends a

marker message over .

When process receives ’s marker message, it records the state of channel

as the single message (five widgets) that it received after it first recorded its state.

The final actual global state is .

The final recorded state is : <$1000, 0>; : <$50, 1995>; : <(five

widgets)>; : < >. Note that this state differs from all the global states through which

the system actually passed.

Termination of the snapshot algorithm • We assume that a process that has received a

marker message records its state within a finite time and sends marker messages over

each outgoing channel within a finite time (even when it no longer needs to send

application messages over these channels). If there is a path of communication channels

and processes from a process to a process , then it is clear on these

assumptions that will record its state a finite time after recorded its state. Since

we are assuming the graph of processes and channels to be strongly connected, it follows

p2

p1

Figure 14.12 The execution of the processes in Figure 14.11

<$1000, 0> p1 (empty) p2 <$50, 2000>

(empty)

c2

c1

1. Global state S0

2. Global state S1

3. Global state S2

4. Global state S3

<$900, 0> p1 (Order 10, $100), M p2 <$50, 2000>

(empty)

c2

c1

<$900, 0> p1 (Order 10, $100), M p2 <$50, 1995>

(five widgets)

c2

c1

<$900, 5> p1 (Order 10, $100) p2 <$50, 1995>

(empty)

c2

c1

(M = marker message)

p1 S0 p1

p1

c

2c

2 S1

p2

c

1 p1 S2

p1 p2 p2

p2

c

2

c

1

p1 p2

c

1

S

3

p1 p2 c1

c

2

pi pj j i ( ) ≠

pj pi634 CHAPTER 14 TIME AND GLOBAL STATES

that all processes will have recorded their states and the states of incoming channels a

finite time after some process initially records its state.

Characterizing the observed state • The snapshot algorithm selects a cut from the history

of the execution. The cut, and therefore the state recorded by this algorithm, is

consistent. To see this, let and be events occurring at and , respectively, such

that . We assert that if is in the cut, then is in the cut. That is, if occurred

before recorded its state, then must have occurred before recorded its state.

This is obvious if the two processes are the same, so we shall assume that . Assume,

for the moment, the opposite of what we wish to prove: that recorded its state before

occurred. Consider the sequence of H messages ( ), giving rise

to the relation . By FIFO ordering over the channels that these messages

traverse, and by the marker sending and receiving rules, a marker message would have

reached ahead of each of . By the marker receiving rule, would

therefore have recorded its state before the event . This contradicts our assumption

that is in the cut, and we are done.

We may further establish a reachability relation between the observed global state

and the initial and final global states when the algorithm runs. Let be

the linearization of the system as it executed (where two events occurred at exactly the

same time, we order them according to process identifiers). Let be the global state

immediately before the first process recorded its state; let be the global state when

the snapshot algorithm terminates, immediately after the last state-recording action; and

let be the recorded global state.

We shall find a permutation of Sys, such that all three states

, and occur in , is reachable from in , and

is reachable from in . Figure 14.13 shows this situation, in which the upper

linearization is Sys and the lower linearization is .

We derive from by first categorizing all events in as pre-snap

events or post-snap events. A pre-snap event at process is one that occurred at

before it recorded its state; all other events are post-snap events. It is important to

understand that a post-snap event may occur before a pre-snap event in , if the events

occur at different processes. (Of course, no post-snap event may occur before a pre-snap

event at the same process.)

We shall show how we may order all pre-snap events before post-snap events to

obtain . Suppose that is a post-snap event at one process, and is a pre-snap

e

i ej pi pj

e

i → ej ej ei ej

pj ei pi

j ≠ i

pi

e

i m1, m2…, mH H ≥ 1

e

i → ej

pj m1, m2…, mH pj

e

j

e

j

Sys e = 0, , e1 …

S

init

S

final

S

snap

Sys′ = e0′ , , , e1′ e2′ …

S

init Ssnap Sfinal Sys′ Ssnap Sinit Sys′ Sfinal

S

snap Sys′

Sys′

Figure 14.13 Reachability between states in the snapshot algorithm

Sinit Sfinal

S

snap

actual execution e0,e1,...

recording recording

begins ends

pre-snap: e'0,e'1,...e'R-1 post-snap: e'R,e'R+1,...

Sys′ Sys Sys

pi pi

Sys

Sys′ ej ej + 1SECTION 14.6 DISTRIBUTED DEBUGGING 635

event at a different process. It cannot be that for then these two events would

be the sending and receiving of a message, respectively. A marker message would have

to have preceded the message, making the reception of the message a post-snap event,

but by assumption is a pre-snap event. We may therefore swap the two events

without violating the happened-before relation (that is, the resultant sequence of events

remains a linearization). The swap does not introduce new process states, since we do

not alter the order in which events occur at any individual process.

We continue swapping pairs of adjacent events in this way as necessary until we

have ordered all pre-snap events prior to all post-snap events

with the resulting execution. For each process, the set of

events in that occurred at it is exactly the set of events that it

experienced before it recorded its state. Therefore the state of each process at that point,

and the state of the communication channels, is that of the global state recorded

by the algorithm. We have disturbed neither of the states or with which the

linearization begins and ends. So we have established the reachability relationship.

Stability and the reachability of the observed state • The reachability property of the

snapshot algorithm is useful for detecting stable predicates. In general, any non-stable

predicate we establish as being True in the state may or may not have been True

in the actual execution whose global state we recorded. However, if a stable predicate is

True in the state then we may conclude that the predicate is True in the state ,

since by definition a stable predicate that is True of a state S is also True of any state

reachable from S. Similarly, if the predicate evaluates to False for , then it must

also be False for .

14.6 Distributed debugging

We now examine the problem of recording a system’s global state so that we may make

useful statements about whether a transitory state – as opposed to a stable state –

occurred in an actual execution. This is what we require, in general, when debugging a

distributed system. We gave an example above in which each of a set of processes

has a variable . The safety condition required in this example is

( ); this constraint is to be met even though a process may change the

value of its variable at any time. Another example is a distributed system controlling a

system of pipes in a factory where we are interested in whether all the valves (controlled

by different processes) were open at some time. In these examples, we cannot in general

observe the values of the variables or the states of the valves simultaneously. The

challenge is to monitor the system’s execution over time – to capture ‘trace’ information

rather than a single snapshot – so that we can establish post hoc whether the required

safety condition was or may have been violated.

Chandy and Lamport’s [1985] snapshot algorithm collects state in a distributed

fashion, and we pointed out how the processes in the system could send the state they

gather to a monitor process for collection. The algorithm we describe next (due to

Marzullo and Neiger [1991]) is centralized. The observed processes send their states to

a process called a monitor, which assembles globally consistent states from what it

receives. We consider the monitor to lie outside the system, observing its execution.

e

j → ej + 1

e

j + 1

e′

0, , , , e1′ e2′ … eR′ – 1

e′

R , , , eR′ + 1 eR′ + 2 … Sys′

e′

0, , , , e1′ e2′ … eR′ – 1

S

snap

S

init Sfinal

S

snap

S

snap Sfinal

S

snap

S

init

pi

x

i xi – xj ≤ δ

i j , = 1 2 , , , … N636 CHAPTER 14 TIME AND GLOBAL STATES

Our aim is to determine cases where a given global state predicate φ was definitely

True at some point in the execution we observed, and cases where it was possibly True.

The notion ‘possibly’ arises as a natural concept because we may extract a consistent

global state S from an executing system and find that is True. No single

observation of a consistent global state allows us to conclude whether a non-stable

predicate ever evaluated to True in the actual execution. Nevertheless, we may be

interested to know whether they might have occurred, as far as we can tell by observing

the execution.

The notion ‘definitely’ does apply to the actual execution and not to a run that we

have extrapolated from it. It may sound paradoxical for us to consider what happened in

an actual execution. However, it is possible to evaluate whether φ was definitely True

by considering all linearizations of the observed events.

We now define the notions of possibly φ and definitely φ for a predicate φ in terms

of linearizations of H, the history of the system’s execution:

possibly φ: The statement possibly φ means that there is a consistent global state S

through which a linearization of H passes such that is True.

definitely φ:The statement definitely φ means that for all linearizations L of H,

there is a consistent global state S through which L passes such that is True.

When we use Chandy and Lamport’s snapshot algorithm and obtain the global state

, we may assert possibly φ if happens to be True. But in general

evaluating possibly φ entails a search through all consistent global states derived from

the observed execution. Only if evaluates to False for all consistent global states

S is it not the case that possibly φ. Note also that while we may conclude definitely

from , we may not conclude from definitely . The latter

is the assertion that holds at some state on every linearization: φ may hold for other

states.

We now describe: how the process states are collected; how the monitor extracts

consistent global states; and how the monitor evaluates possibly φ and definitely φ in

both asynchronous and synchronous systems.

14.6.1 Collecting the state

The observed processes send their initial state to the monitor

initially, and thereafter from time to time, in state messages. The monitor records the

state messages from each process in a separate queue , for each .

The activity of preparing and sending state messages may delay the normal

execution of the observed processes, but it does not otherwise interfere with it. There is

no need to send the state except initially and when it changes. There are two

optimizations to reduce the state-message traffic to the monitor. First, the global state

predicate may depend only on certain parts of the processes’ states – for example, only

on the states of particular variables – so the observed processes need only send the

relevant state to the monitor. Second, they need only send their state at times when the

predicate φ may become True or cease to be True. There is no point in sending changes

to the state that do not affect the predicate’s value.

φ( ) S

φ( ) S

φ( ) S

S

snap φ( ) Ssnap

φ( ) S

( ) ¬φ

¬possibly φ ¬ φ possibly φ ( ) ¬

¬φ

pi i ( ) = 1 2 , , , … N

pi Qi i = 1 2 , , , … NSECTION 14.6 DISTRIBUTED DEBUGGING 637

For example, in the example system of processes that are supposed to obey the

constraint . ( ), each process need only notify the monitor

when the values of its own variable changes. When they send their state, they supply

the value of but do not need to send any other variables.

14.6.2 Observing consistent global states

The monitor must assemble consistent global states against which it evaluates φ. Recall

that a cut C is consistent if and only if for all events e in the cut C, .

For example, Figure 14.14 shows two processes and with variables and

, respectively. The events shown on the timelines (with vector timestamps) are

adjustments to the values of the two variables. Initially, . The requirement

is . The processes make adjustments to their variables, but ‘large’

adjustments cause a message containing the new value to be sent to the other process.

When either of the processes receives an adjustment message from the other, it sets its

variable equal to the value contained in the message.

Whenever one of the processes or adjusts the value of its variable (whether

it is a ‘small’ adjustment or a ‘large’ one), it sends the value in a state message to the

monitor. The latter keeps the state messages in the per-process queues for analysis. If the

monitor were to use values from the inconsistent cut in Figure 14.14, then it would

find that , breaking the constraint . But this state of

affairs never occurred. On the other hand, values from the consistent cut show

.

In order that the monitor can distinguish consistent global states from inconsistent

global states, the observed processes enclose their vector clock values with their state

messages. Each queue is kept in sending order, which can immediately be

established by examining the ith component of the vector timestamps. Of course, the

monitor may deduce nothing about the ordering of states sent by different processes

from their arrival order, because of variable message latencies. It must instead examine

the vector timestamps of the state messages.

Let be a global state drawn from the state messages that the

monitor has received. Let be the vector timestamp of the state received from

. Then it can be shown that S is a consistent global state if and only if:

pi

x

i – xj ≤ δ i j , = 1 2 , , , … N

x

i

x

i

f e →  f C ∈

Figure 14.14 Vector timestamps and variable values for the execution of Figure 14.9

m1 m2

p1

p2 Physical

time

Cut C1

(1,0) (2,0) (4,3)

(2,1) (2,2) (2,3)

(3,0)

x1= 1 x1= 100 x1= 105

x2= 100 x2= 95 x2= 90

x1= 90

Cut C2

p1 p2 x1

x

2

x

1 = = x2 0

x

1 – 50 x2 ≤

p1 p2

C

1

x

1 = = 1, x2 100 x1 – 50 x2 ≤

C

2

x

1 = = 105, x2 90

Qi

S s

= ( ) 1, , , s2 … sN

V s

( ) i si

pi638 CHAPTER 14 TIME AND GLOBAL STATES

for – (Condition CGS)

This says that the number of ’s events known at when it sent is no more than

the number of events that had occurred at when it sent . In other words, if one

process’s state depends upon another (according to happened-before ordering), then the

global state also encompasses the state upon which it depends.

In summary, we now possess a method whereby the monitor may establish

whether a given global state is consistent, using the vector timestamps kept by the

observed processes and piggybacked on the state messages that they send to it.

Figure 14.15 shows the lattice of consistent global states corresponding to the

execution of the two processes in Figure 14.14. This structure captures the relation of

reachability between consistent global states. The nodes denote global states, and the

edges denote possible transitions between these states. The global state has both

processes in their initial state; has still in its initial state and in the next state

in its local history. The state is not consistent, because of the message sent from

to , so it does not appear in the lattice.

The lattice is arranged in levels with, for example, in level 0 and in level

1. In general, is in level . A linearization traverses the lattice from any global

state to any global state reachable from it on the next level – that is, in each step some

process experiences one event. For example, is reachable from , but is not

reachable from .

The lattice shows us all the linearizations corresponding to a history. It is now

clear in principle how a monitor should evaluate possibly φ and definitely φ. To evaluate

possibly φ, the monitor starts at the initial state and steps through all consistent states

reachable from that point, evaluating φ at each stage. It stops when φ evaluates to True.

To evaluate definitely φ, the monitor must attempt to find a set of states through which

all linearizations must pass, and at each of which φ evaluates to True. For example, if

and in Figure 14.15 are both True then, since all linearizations pass

through these states, definitely φ holds.

V s

( ) i [ ] i ≥ V s ( ) j [ ] i i j , = 1 2 , , , … N

pi pj sj

pi si

Figure 14.15 The lattice of global states for the execution of Figure 14.14

Sij= global state after i events at process 1

and j events at process 2

S00

S10

S20

S30 S21

S31

S32

S22

S23

S33

S43

Level 0

1 2 3 4 5 6 7

S

00

S

10 p2 p1

S

01 m1

p1 p2

S

00 S10

S

ij ( ) i j +

S

22 S20 S22

S

30

φ( ) φ S30 ( ) S21SECTION 14.6 DISTRIBUTED DEBUGGING 639

14.6.3 Evaluating possibly φ

To evaluate possibly φ, the monitor must traverse the lattice of reachable states, starting

from the initial state . The algorithm is shown in Figure 14.16. The

algorithm assumes that the execution is infinite. It may easily be adapted for a finite

execution.

The monitor may discover the set of consistent states in level reachable

from a given consistent state in level L by the following method. Let

be a consistent state. Then a consistent state in the next level

reachable from S is of the form , which differs from S only by

containing the next state (after a single event) of some process . The monitor can find

all such states by traversing the queues of state messages ( ). The

state is reachable from S if and only if:

for :

This condition comes from condition CGS above and from the fact that S was already a

consistent global state. A given state may in general be reached from several states at the

previous level, so the monitor should take care to evaluate the consistency of each state

only once.

s

01

s

02

…s

0N

( ) ,

Figure 14.16 Algorithms to evaluate possibly φ and definitely φ

1. Evaluating possibly φ for global history H of N processes

L := 0;

States := { };

while ( for all )

L := L + 1;

Reachable := { : reachable in H from some };

States := Reachable

end while

output "possibly φ";

2. Evaluating definitely φ for global history H of N processes

L := 0;

if (φ ) then States := {} else States := { };

while (States  {})

L := L + 1;

Reachable := { : reachable in H from some };

States := { : }

end while

output "definitely φ";

s

01

s

02

… s

0N

( ) , , ,

φ( ) S = False S States ∈

S′ S′ S States ∈ ∧ level S ( ) ′ = L

s

01

s

02

… s

0N

( ) , , , ( ) s1 0, , , s2 0 … sN 0

S′ S′ S States ∈ ∧ level S ( ) ' = L

S Reachable ∈ φ( ) S = False

L + 1

S s

= ( ) 1, , , s2 … sN

S′ s

= ( ) 1, , , s2 …si′ …, sN

pi

Qi i = 1 2 , , , … N

S′

j = 1 2 , , , … N j i , ≠ V s ( ) j [ ] j ≥ V s ( ) i′ [ ] j640 CHAPTER 14 TIME AND GLOBAL STATES

14.6.4 Evaluating definitely φ

To evaluate definitely φ, the monitor again traverses the lattice of reachable states a level

at a time, starting from the initial state . The algorithm (shown in Figure

14.16) again assumes that the execution is infinite but may easily be adapted for a finite

execution. It maintains the set States, which contains those states at the current level that

may be reached on a linearization from the initial state by traversing only states for

which φ evaluates to False. As long as such a linearization exists, we may not assert

definitely φ: the execution could have taken this linearization, and φ would be False at

every stage along it. If we reach a level for which no such linearization exists, we may

conclude definitely φ.

In Figure 14.17, at level 3 the set States consists of only one state, which is

reachable by a linearization on which all states are False (marked in bold lines). The

only state considered at level 4 is the one marked ‘F’. (The state to its right is not

considered, since it can only be reached via a state for which φ evaluates to True.) If φ

evaluates to True in the state at level 5, then we may conclude definitely φ. Otherwise,

the algorithm must continue beyond this level.

Cost • The algorithms we have just described are combinatorially explosive. Suppose

that k is the maximum number of events at a single process. Then the algorithms we have

described entail comparisons (the monitor compares the states of each of the N

observed processes with one another).

There is also a space cost to these algorithms of . However, we observe

that the monitor may delete a message containing state from queue when no other

item of state arriving from another process could possibly be involved in a consistent

global state containing . That is, when:

for

where is the last state that the monitor has received from process .

s

01

s

02

… s

0N

( ) , , ,

Figure 14.17 Evaluating definitely φ

F = (φ(S) = False); T = (φ(S) = True)

?

–

Level 0

1 2 3 4 5

F

F

F

F T

F

O k ( ) N

O kN ( )

s

i Qi

s

i

V s

j

last

( )[ ] i > V s ( ) i [ ] i j = 1 2 , , , , … N j i ≠

s

j

last

pjSECTION 14.6 DISTRIBUTED DEBUGGING 641

14.6.5 Evaluating possibly φ and definitely φ in synchronous systems

The algorithms we have given so far work in an asynchronous system: we have made no

timing assumptions. But the price paid for this is that the monitor may examine a

consistent global state for which any two local states and

occurred an arbitrarily long time apart in the actual execution of the system. Our

requirement, by contrast, is to consider only those global states that the actual execution

could in principle have traversed.

In a synchronous system, suppose that the processes keep their physical clocks

internally synchronized within a known bound, and that the observed processes provide

physical timestamps as well as vector timestamps in their state messages. Then the

monitor need consider only those consistent global states whose local states could

possibly have existed simultaneously, given the approximate synchronization of the

clocks. With good enough clock synchronization, these will number many less than all

globally consistent states.

We now give an algorithm to exploit synchronized clocks in this way. We assume

that each observed process ( ) and the monitor, which we shall call

, keep a physical clock ( ). These are synchronized to within a

known bound ; that is, at the same real time:

for

The observed processes send both their vector time and physical time with their state

messages to the monitor. The monitor now applies a condition that not only tests for

consistency of a global state , but also tests whether each pair of

states could have happened at the same real time, given the physical clock values. In

other words, for :

and and could have occurred at the same real time.

The first clause is the condition that we used earlier. For the second clause, note that

is in the state from the time it first notifies the monitor, , to some later local

time – say, when the next state transition occurs at . For and to have

obtained at the same real time we thus have, allowing for the bound on clock

synchronization:

– or vice versa (swapping i and j).

The monitor must calculate a value for , which is measured against ’s clock. If

the monitor has received a state message for ’s next state , then is .

Otherwise, the monitor estimates as – max + D, where is the monitor’s

current local clock value and max is the maximum transmission time for a state message.

S s

= ( ) 1, , , s2 … sN si sj

pi i = 1 2 , , , … N

p0 Ci i = 0 1 , , , … N

D > 0

C

i( ) t – Cj( ) t < D i j , = 0 1 , , , … N

S s

= ( ) 1, , , s2 … sN

i j , = 1 2 , , , … N

V s

( ) i [ ] i ≥ V s ( ) j [ ] i si sj

pi

s

i Ci( ) si

L

i( ) si pi si sj

C

i( ) si – D C ≤ j( ) sj ≤ Li( ) si + D

L

i( ) si pi

pi si′ Li( ) si Ci( ) si′

L

i( ) si C0 C0642 CHAPTER 14 TIME AND GLOBAL STATES

14.7 Summary

This chapter began by describing the importance of accurate timekeeping for distributed

systems. It then described algorithms for synchronizing clocks despite the drift between

them and the variability of message delays between computers.

The degree of synchronization accuracy that is practically obtainable fulfils many

requirements but is nonetheless not sufficient to determine the ordering of an arbitrary

pair of events occurring at different computers. The happened-before relation is a partial

order on events that reflects a flow of information between them – within a process, or

via messages between processes. Some algorithms require events to be ordered in

happened-before order, for example, successive updates made to separate copies of data.

Lamport clocks are counters that are updated in accordance with the happened-before

relationship between events. Vector clocks are an improvement on Lamport clocks, in

that it is possible to determine by examining their vector timestamps whether two events

are ordered by happened-before or are concurrent.

We introduced the concepts of events, local and global histories, cuts, local and

global states, runs, consistent states, linearizations (consistent runs) and reachability. A

consistent state or run is one that is in accord with the happened-before relation.

We went on to consider the problem of recording a consistent global state by

observing a system’s execution. Our objective was to evaluate a predicate on this state.

An important class of predicates are the stable predicates. We described the snapshot

algorithm of Chandy and Lamport, which captures a consistent global state and allows

us to make assertions about whether a stable predicate holds in the actual execution. We

went on to give Marzullo and Neiger’s algorithm for deriving assertions about whether

a predicate held or may have held in the actual run. This algorithm employs a monitor

process to collect states. The monitor examines vector timestamps to extract consistent

global states, and it constructs and examines the lattice of all consistent global states.

This algorithm involves great computational complexity but is valuable for

understanding and can be of some practical benefit in real systems where relatively few

events change the global predicate’s value. The algorithm has a more efficient variant in

synchronous systems, where clocks may be synchronized.EXERCISES 643

EXERCISES

14.1 What is clock skew? Also, explain the concepts of International Atomic Time and

Coordinated Universal Time. pages 614, 615

14.2 A clock is reading 11:34:26.0 (hr:min:sec) when it is discovered to be 6 seconds fast.

Explain why it is undesirable to set it back to the right time at that point, and show

(numerically) how it should be adjusted so as to be correct after 12 seconds have

elapsed. page 616

14.3 A scheme for implementing at-most-once reliable message delivery uses synchronized

clocks to reject duplicate messages. Processes place their local clock value (a

‘timestamp’) in the messages they send. Each receiver keeps a table giving, for each

sending process, the largest message timestamp it has seen. Assume that clocks are

synchronized to within 100 ms, and that messages can arrive at most 50 ms after

transmission.

i) When may a process ignore a message bearing a timestamp T, if it has recorded

the last message received from that process as having timestamp ?

ii) When may a receiver remove a timestamp 175,000 (ms) from its table? (Hint: use

the receiver’s local clock value.)

iii) Should the clocks be internally synchronized or externally synchronized?

page 617

14.4 A client attempts to synchronize with a time server. It records the round-trip times and

timestamps returned by the server in the table below.

Which of these times should it use to set its clock? To what time should it set it?

Estimate the accuracy of the setting with respect to the server’s clock. If it is known that

the time between sending and receiving a message in the system concerned is at least 8

ms, do your answers change?

page 617

14.5 Synchronization of clocks is necessary for communication between nodes. Explain the

two modes of synchronization. page 615

14.6 What is the main disadvantage of Cristian’s method? page 618

14.7 An NTP server B receives server A’s message at 18:47:56.76, bearing a timestamp

18:46:33.46, and replies to it. A receives the message at 18:49:15.72, bearing B’s

timestamp, 18:48:25.72. Estimate the offset between B and A, and the accuracy of the

estimate. page 621

T′

Round-trip (ms) Time (hr:min:sec)

22 10:54:23.674

25 10:54:25.450

20 10:54:28.342644 CHAPTER 14 TIME AND GLOBAL STATES

14.8 How many modes do NTP servers use to synchronize with one another? Describe each

mode. page 621

14.9 Lamport stated that clocks cannot be synchronized perfectly across a distributed system.

In accordance with this statement, what action can be taken? page 623

14.10 By considering a chain of zero or more messages connecting events e and and using

induction, show that . page 624

14.11 What is total ordering of a logical clock? Explain with an example. page 625

14.12 In a similar fashion to Exercise 14.10, show that . page 626

14.13 Using the result of Exercise 14.11, show that if events e and are concurrent then

neither nor . Hence show that if then .

page 626

14.14 Two processes P and Q are connected in a ring using two channels, and they constantly

rotate a message m. At any one time, there is only one copy of m in the system. Each

process’s state consists of the number of times it has received m, and P sends m first. At

a certain point, P has the message and its state is 101. Immediately after sending m, P

initiates the snapshot algorithm. Explain the operation of the algorithm in this case,

giving the possible global state(s) reported by it. page 631

14.15 The figure above shows events occurring for each of two processes, p1 and p2. Arrows

between processes denote message transmission.

Draw and label the lattice of consistent states (p1 state, p2 state), beginning with

the initial state (0,0).

page 638

14.16 Jones is running a collection of processes . Each process contains a

variable . She wishes to determine whether all the variables were ever

equal in the course of the execution.

i) Jones’ processes run in a synchronous system. She uses a monitor process to

determine whether the variables were ever equal. When should the application

processes communicate with the monitor process, and what should their messages

contain?

ii) Explain the statement possibly ( ). How can Jones determine

whether this statement is true of her execution?

page 639

e′

e e → ′  L e ( ) < L e ( ) ′

e e → ′  V e ( ) < V e ( ) ′

e′

V e ( ) ≤ V e ( ) ′ V e ( ) ′ ≤ V e ( ) V e ( ) < V e ( ) ′ e e → ′

Time

p1

p2

p1, , , p2 … pN pi

v

i v1, , , v2 … vN

v

1 = = = v2 … vN645

15

COORDINATION AND AGREEMENT

15.1 Introduction

15.2 Distributed mutual exclusion

15.3 Elections

15.4 Coordination and agreement in group communication

15.5 Consensus and related problems

15.6 Summary

In this chapter, we introduce some topics and algorithms related to the issue of how

processes coordinate their actions and agree on shared values in distributed systems,

despite failures. The chapter begins with algorithms to achieve mutual exclusion among

a collection of processes, so as to coordinate their accesses to shared resources. It goes

on to examine how an election can be implemented in a distributed system – that is, how

a group of processes can agree on a new coordinator of their activities after the previous

coordinator has failed.

The second half of the chapter examines the related problems of group

communication, consensus, Byzantine agreement and interactive consistency. In the

context of group communication, the issue is how to agree on such matters as the order

in which messages are to be delivered. Consensus and the other problems generalize from

this: how can any collection of processes agree on some value, no matter what the domain

of the values in question? We encounter a fundamental result in the theory of distributed

systems: that under certain conditions – including surprisingly benign failure conditions

– it is impossible to guarantee that processes will reach consensus.646 CHAPTER 15 COORDINATION AND AGREEMENT

15.1 Introduction

This chapter introduces a collection of algorithms whose goals vary but that share an aim

that is fundamental in distributed systems: for a set of processes to coordinate their

actions or to agree on one or more values. For example, in the case of a complex piece

of machinery such as a spaceship, it is essential that the computers controlling it agree

on such conditions as whether the spaceship’s mission is proceeding or has been

aborted. Furthermore, the computers must coordinate their actions correctly with respect

to shared resources (the spaceship’s sensors and actuators). The computers must be able

to do so even where there is no fixed master-slave relationship between the components

(which would make coordination particularly simple). The reason for avoiding fixed

master-slave relationships is that we often require our systems to keep working correctly

even if failures occur, so we need to avoid single points of failure, such as fixed masters.

An important distinction for us, as in Chapter 14, will be whether the distributed

system under study is asynchronous or synchronous. In an asynchronous system we can

make no timing assumptions. In a synchronous system, we shall assume that there are

bounds on the maximum message transmission delay, on the time taken to execute each

step of a process, and on clock drift rates. The synchronous assumptions allow us to use

timeouts to detect process crashes.

Another important aim of the chapter is to consider failures, and how to deal with

them when designing algorithms. Section 2.4.2 introduced a failure model, which we

shall use in this chapter. Coping with failures is a subtle business, so we begin by

considering some algorithms that tolerate no failures and progress through benign

failures before exploring how to tolerate arbitrary failures. Along the way, we encounter

a fundamental result in the theory of distributed systems: even under surprisingly benign

failure conditions, it is impossible to guarantee in an asynchronous system that a

collection of processes can agree on a shared value – for example, for all of a spaceship’s

controlling processes to agree ‘mission proceed’ or ‘mission abort’.

Section 15.2 examines the problem of distributed mutual exclusion. This is the

extension to distributed systems of the familiar problem of avoiding race conditions in

kernels and multi-threaded applications. Since much of what occurs in distributed

systems is resource sharing, this is an important problem to solve. Next, Section 15.3

introduces the related but more general issue of how to ‘elect’ one of a collection of

processes to perform a special role. For example, in Chapter 14 we saw how processes

synchronize their clocks to a designated time server. If this server fails and several

surviving servers can fulfil that role, then for the sake of consistency it is necessary to

choose just one server to take over.

Coordination and agreement related to group communication is the subject of

Section 15.4. As Section 4.4.1 explained, the ability to multicast a message to a group is

a very useful communication paradigm, with applications from locating resources to

coordinating the updates to replicated data. Section 15.4 examines multicast reliability

and ordering semantics, and gives algorithms to achieve the variations. Multicast

delivery is essentially a problem of agreement between processes: the recipients agree

on which messages they will receive, and in which order they will receive them. Section

15.5 discusses the problem of agreement more generally, primarily in the forms known

as consensus and Byzantine agreement.SECTION 15.1 INTRODUCTION 647

The treatment followed in this chapter involves stating the assumptions and the

goals to be met, and giving an informal account of why the algorithms presented are

correct. There is insufficient space to provide a more rigorous approach. For that, we

refer the reader to a text that gives a thorough account of distributed algorithms, such as

Attiya and Welch [1998] and Lynch [1996].

Before presenting the problems and algorithms, we discuss failure assumptions

and the practical matter of detecting failures in distributed systems.

15.1.1 Failure assumptions and failure detectors

For the sake of simplicity, this chapter assumes that each pair of processes is connected

by reliable channels. That is, although the underlying network components may suffer

failures, the processes use a reliable communication protocol that masks these failures –

for example, by retransmitting missing or corrupted messages. Also for the sake of

simplicity, we assume that no process failure implies a threat to the other processes’

ability to communicate. This means that none of the processes depends upon another to

forward messages.

Note that a reliable channel eventually delivers a message to the recipient’s input

buffer. In a synchronous system, we suppose that there is hardware redundancy where

necessary, so that a reliable channel not only eventually delivers each message despite

underlying failures, but does so within a specified time bound.

In any particular interval of time, communication between some processes may

succeed while communication between others is delayed. For example, the failure of a

router between two networks may mean that a collection of four processes is split into

two pairs, such that intra-pair communication is possible over their respective networks;

but inter-pair communication is not possible while the router has failed. This is known

as a network partition (Figure 15.1). Over a point-to-point network such as the Internet,

complex topologies and independent routing choices mean that connectivity may be

asymmetric: communication is possible from process p to process q, but not vice versa.

Connectivity may also be intransitive: communication is possible from p to q and from

q to r, but p cannot communicate directly with r. Thus our reliability assumption entails

that eventually any failed link or router will be repaired or circumvented. Nevertheless,

the processes may not all be able to communicate at the same time.

Figure 15.1 A network partition

Crashed

router648 CHAPTER 15 COORDINATION AND AGREEMENT

The chapter assumes, unless we state otherwise, that processes fail only by

crashing – an assumption that is good enough for many systems. In Section 15.5, we

shall consider how to treat the cases where processes have arbitrary (Byzantine) failures.

Whatever the type of failure, a correct process is one that exhibits no failures at any

point in the execution under consideration. Note that correctness applies to the whole

execution, not just to a part of it. So a process that suffers a crash failure is ‘non-failed’

before that point, not ‘correct’ before that point.

One of the problems in the design of algorithms that can overcome process crashes

is that of deciding when a process has crashed. A failure detector [Chandra and Toueg

1996, Stelling et al. 1998] is a service that processes queries about whether a particular

process has failed. It is often implemented by an object local to each process (on the

same computer) that runs a failure-detection algorithm in conjunction with its

counterparts at other processes. The object local to each process is called a local failure

detector. We outline how to implement failure detectors shortly, but first we concentrate

on some of the properties of failure detectors.

A failure ‘detector’ is not necessarily accurate. Most fall into the category of

unreliable failure detectors. An unreliable failure detector may produce one of two

values when given the identity of a process: Unsuspected or Suspected. Both of these

results are hints, which may or may not accurately reflect whether the process has

actually failed. A result of Unsuspected signifies that the detector has recently received

evidence suggesting that the process has not failed; for example, a message was recently

received from it. But of course, the process may have failed since then. A result of

Suspected signifies that the failure detector has some indication that the process may

have failed. For example, it may be that no message from the process has been received

for more than a nominal maximum length of silence (even in an asynchronous system,

practical upper bounds can be used as hints). The suspicion may be misplaced: for

example, the process could be functioning correctly but be on the other side of a network

partition, or it could be running more slowly than expected.

A reliable failure detector is one that is always accurate in detecting a process’s

failure. It answers processes’ queries with either a response of Unsuspected – which, as

before, can only be a hint – or Failed. A result of Failed means that the detector has

determined that the process has crashed. Recall that a process that has crashed stays that

way, since by definition a process never takes another step once it has crashed.

It is important to realize that, although we speak of one failure detector acting for

a collection of processes, the response that the failure detector gives to a process is only

as good as the information available at that process. A failure detector may sometimes

give different responses to different processes, since communication conditions vary

from process to process.

We can implement an unreliable failure detector using the following algorithm.

Each process p sends a ‘p is here’ message to every other process, and it does this every

T seconds. The failure detector uses an estimate of the maximum message transmission

time of D seconds. If the local failure detector at process q does not receive a ‘p is here’

message within seconds of the last one, then it reports to q that p is Suspected.

However, if it subsequently receives a ‘p is here’ message, then it reports to q that p is

OK.

In a real distributed system, there are practical limits on message transmission

times. Even email systems give up after a few days, since it is likely that communication

T D +SECTION 15.2 DISTRIBUTED MUTUAL EXCLUSION 649

links and routers will have been repaired in that time. If we choose small values for T

and D (so that they total 0.1 second, say), then the failure detector is likely to suspect

non-crashed processes many times, and much bandwidth will be taken up with ‘p is

here’ messages. If we choose a large total timeout value (a week, say), then crashed

processes will often be reported as Unsuspected.

A practical solution to this problem is to use timeout values that reflect the

observed network delay conditions. If a local failure detector receives a ‘p is here’ in 20

seconds instead of the expected maximum of 10 seconds, it can reset its timeout value

for p accordingly. The failure detector remains unreliable, and its answers to queries are

still only hints, but the probability of its accuracy increases.

In a synchronous system, our failure detector can be made into a reliable one. We

can choose D so that it is not an estimate but an absolute bound on message transmission

times; the absence of a ‘p is here’ message within seconds entitles the local

failure detector to conclude that p has crashed.

The reader may wonder whether failure detectors are of any practical use.

Unreliable failure detectors may suspect a process that has not failed (they may be

inaccurate), and they may not suspect a process that has in fact failed (they may be

incomplete). Reliable failure detectors, on the other hand, require that the system is

synchronous (and few practical systems are).

We have introduced failure detectors because they help us to think about the

nature of failures in a distributed system. And any practical system that is designed to

cope with failures must detect them – however imperfectly. But it turns out that even

unreliable failure detectors with certain well-defined properties can help us to provide

practical solutions to the problem of coordinating processes in the presence of failures.

We return to this point in Section 15.5.

15.2 Distributed mutual exclusion

Distributed processes often need to coordinate their activities. If a collection of

processes share a resource or collection of resources, then often mutual exclusion is

required to prevent interference and ensure consistency when accessing the resources.

This is the critical section problem, familiar in the domain of operating systems. In a

distributed system, however, neither shared variables nor facilities supplied by a single

local kernel can be used to solve it, in general. We require a solution to distributed

mutual exclusion: one that is based solely on message passing.

In some cases shared resources are managed by servers that also provide

mechanisms for mutual exclusion – Chapter 16 describes how some servers synchronize

client accesses to resources. But in some practical cases, a separate mechanism for

mutual exclusion is required.

Consider users who update a text file. A simple means of ensuring that their

updates are consistent is to allow them to access it only one at a time, by requiring the

editor to lock the file before updates can be made. NFS file servers, described in Chapter

12, are designed to be stateless and therefore do not support file locking. For this reason,

UNIX systems provide a separate file-locking service, implemented by the daemon

lockd, to handle locking requests from clients.

T D +650 CHAPTER 15 COORDINATION AND AGREEMENT

A particularly interesting example is where there is no server, and a collection of

peer processes must coordinate their accesses to shared resources amongst themselves.

This occurs routinely on networks such as Ethernets and IEEE 802.11 wireless networks

in ‘ad hoc’ mode, where network interfaces cooperate as peers so that only one node

transmits at a time on the shared medium. Consider, also, a system monitoring the

number of vacancies in a car park with a process at each entrance and exit that tracks the

number of vehicles entering and leaving. Each process keeps a count of the total number

of vehicles within the car park and displays whether or not it is full. The processes must

update the shared count of the number of vehicles consistently. There are several ways

of achieving that, but it would be convenient for these processes to be able to obtain

mutual exclusion solely by communicating among themselves, eliminating the need for

a separate server.

It is useful to have a generic mechanism for distributed mutual exclusion at our

disposal – one that is independent of the particular resource management scheme in

question. We now examine some algorithms for achieving that.

15.2.1 Algorithms for mutual exclusion

We consider a system of N processes , that do not share variables.

The processes access common resources, but they do so in a critical section. For the sake

of simplicity, we assume that there is only one critical section. It is straightforward to

extend the algorithms we present to more than one critical section.

We assume that the system is asynchronous, that processes do not fail and that

message delivery is reliable, so that any message sent is eventually delivered intact,

exactly once.

The application-level protocol for executing a critical section is as follows:

enter() // enter critical section – block if necessary

resourceAccesses() // access shared resources in critical section

exit() // leave critical section – other processes may now enter

Our essential requirements for mutual exclusion are as follows:

ME1: (safety) At most one process may execute in the critical section

(CS) at a time.

ME2: (liveness) Requests to enter and exit the critical section eventually

succeed.

Condition ME2 implies freedom from both deadlock and starvation. A deadlock would

involve two or more of the processes becoming stuck indefinitely while attempting to

enter or exit the critical section, by virtue of their mutual interdependence. But even

without a deadlock, a poor algorithm might lead to starvation: the indefinite

postponement of entry for a process that has requested it.

The absence of starvation is a fairness condition. Another fairness issue is the

order in which processes enter the critical section. It is not possible to order entry to the

critical section by the times that the processes requested it, because of the absence of

global clocks. But a useful fairness requirement that is sometimes made makes use of

the happened-before ordering (Section 14.4) between messages that request entry to the

critical section:

pi, i = 1 2 , , , … NSECTION 15.2 DISTRIBUTED MUTUAL EXCLUSION 651

ME3: ( ordering) If one request to enter the CS happened-before another,

then entry to the CS is granted in that order.

If a solution grants entry to the critical section in happened-before order, and if all

requests are related by happened-before, then it is not possible for a process to enter the

critical section more than once while another waits to enter. This ordering also allows

processes to coordinate their accesses to the critical section. A multi-threaded process

may continue with other processing while a thread waits to be granted entry to a critical

section. During this time, it might send a message to another process, which

consequently also tries to enter the critical section. ME3 specifies that the first process

be granted access before the second.

We evaluate the performance of algorithms for mutual exclusion according to the

following criteria:

• the bandwidth consumed, which is proportional to the number of messages sent in

each entry and exit operation;

• the client delay incurred by a process at each entry and exit operation;

• the algorithm’s effect upon the throughput of the system. This is the rate at which

the collection of processes as a whole can access the critical section, given that

some communication is necessary between successive processes. We measure the

effect using the synchronization delay between one process exiting the critical

section and the next process entering it; the throughput is greater when the

synchronization delay is shorter.

We do not take the implementation of resource accesses into account in our descriptions.

We do, however, assume that the client processes are well behaved and spend a finite

time accessing resources within their critical sections.

The central server algorithm • The simplest way to achieve mutual exclusion is to

employ a server that grants permission to enter the critical section. Figure 15.2 shows

the use of this server. To enter a critical section, a process sends a request message to

→

Figure 15.2 Server managing a mutual exclusion token for a set of processes

Server

1. Request

token

Queue of

requests

2. Release

token

3. Grant

token

p 4

p

p 2 3

p1

4 2652 CHAPTER 15 COORDINATION AND AGREEMENT

the server and awaits a reply from it. Conceptually, the reply constitutes a token

signifying permission to enter the critical section. If no other process has the token at the

time of the request, then the server replies immediately, granting the token. If the token

is currently held by another process, then the server does not reply, but queues the

request. When a process exits the critical section, it sends a message to the server, giving

it back the token.

If the queue of waiting processes is not empty, then the server chooses the oldest

entry in the queue, removes it and replies to the corresponding process. The chosen

process then holds the token. In the figure, we show a situation in which ’s request

has been appended to the queue, which already contained ’s request. exits the

critical section, and the server removes ’s entry and grants permission to enter to

by replying to it. Process does not currently require entry to the critical section.

Given our assumption that no failures occur, it is easy to see that the safety and

liveness conditions are met by this algorithm. The reader should verify, however, that

the algorithm does not satisfy property ME3.

We now evaluate the performance of this algorithm. Entering the critical section

– even when no process currently occupies it – takes two messages (a request followed

by a grant) and delays the requesting process by the time required for this round-trip.

Exiting the critical section takes one release message. Assuming asynchronous message

passing, this does not delay the exiting process.

The server may become a performance bottleneck for the system as a whole. The

synchronization delay is the time taken for a round-trip: a release message to the server,

followed by a grant message to the next process to enter the critical section.

A ring-based algorithm • One of the simplest ways to arrange mutual exclusion between

the N processes without requiring an additional process is to arrange them in a logical

ring. This requires only that each process has a communication channel to the next

process in the ring, . The idea is that exclusion is conferred by obtaining a

token in the form of a message passed from process to process in a single direction –

p2

p4 p3

p4 p4

p1

pn

p2

p3 p4

Token

Figure 15.3 A ring of processes transferring a mutual exclusion token

p1

pi

p( ) i + 1 mod NSECTION 15.2 DISTRIBUTED MUTUAL EXCLUSION 653

clockwise, say – around the ring. The ring topology may be unrelated to the physical

interconnections between the underlying computers.

If a process does not require to enter the critical section when it receives the token,

then it immediately forwards the token to its neighbour. A process that requires the token

waits until it receives it, but retains it. To exit the critical section, the process sends the

token on to its neighbour.

The arrangement of processes is shown in Figure 15.3. It is straightforward to

verify that the conditions ME1 and ME2 are met by this algorithm, but that the token is

not necessarily obtained in happened-before order. (Recall that the processes may

exchange messages independently of the rotation of the token.)

This algorithm continuously consumes network bandwidth (except when a

process is inside the critical section): the processes send messages around the ring even

when no process requires entry to the critical section. The delay experienced by a

process requesting entry to the critical section is between 0 messages (when it has just

received the token) and N messages (when it has just passed on the token). To exit the

critical section requires only one message. The synchronization delay between one

process’s exit from the critical section and the next process’s entry is anywhere from 1

to N message transmissions.

An algorithm using multicast and logical clocks • Ricart and Agrawala [1981] developed an algorithm to implement mutual exclusion between N peer processes that is

based upon multicast. The basic idea is that processes that require entry to a critical section multicast a request message, and can enter it only when all the other processes have

Figure 15.4 Ricart and Agrawala’s algorithm

On initialization

state := RELEASED;

To enter the section

state := WANTED;

Multicast request to all processes;

T := request’s timestamp;

Wait until (number of replies received = (N – 1));

state := HELD;

On receipt of a request <Ti, pi> at pj (i j)

if (state = HELD or (state = WANTED and (T, pj) < (Ti, pi)))

then

queue request from pi without replying;

else

reply immediately to pi;

end if

To exit the critical section

state := RELEASED;

reply to any queued requests;

Request processing deferred here654 CHAPTER 15 COORDINATION AND AGREEMENT

replied to this message. The conditions under which a process replies to a request are

designed to ensure that conditions ME1–ME3 are met.

The processes bear distinct numeric identifiers. They are assumed

to possess communication channels to one another, and each process keeps a

Lamport clock, updated according to the rules LC1 and LC2 of Section 14.4. Messages

requesting entry are of the form < >, where T is the sender’s timestamp and is

the sender’s identifier.

Each process records its state of being outside the critical section (RELEASED),

wanting entry (WANTED) or being in the critical section (HELD) in a variable state. The

protocol is given in Figure 15.4.

If a process requests entry and the state of all other processes is RELEASED, then

all processes will reply immediately to the request and the requester will obtain entry. If

some process is in the state HELD, then that process will not reply to requests until it has

finished with the critical section, and so the requester cannot gain entry in the meantime.

If two or more processes request entry at the same time, then whichever process’s

request bears the lowest timestamp will be the first to collect replies, granting it

entry next. If the requests bear equal Lamport timestamps, the requests are ordered

according to the processes’ corresponding identifiers. Note that, when a process requests

entry, it defers processing requests from other processes until its own request has been

sent and it has recorded the timestamp T of the request. This is so that processes make

consistent decisions when processing requests.

This algorithm achieves the safety property ME1. If it were possible for two

processes and ( ) to enter the critical section at the same time, then both of

those processes would have to have replied to the other. But since the pairs < > are

totally ordered, this is impossible. We leave the reader to verify that the algorithm also

meets requirements ME2 and ME3.

To illustrate the algorithm, consider a situation involving three processes, ,

and , shown in Figure 15.5. Let us assume that is not interested in entering the

critical section, and that and request entry concurrently. The timestamp of ’s

request is 41, and that of is 34. When receives their requests, it replies

p1, , , p2 … pN

pi

T p , i pi

N – 1

pi pj i j ≠

Ti

, pi

p1 p2

p3

Figure 15.5 Multicast synchronization

p3

34

Reply

34

41

41

41

34

p1

p2

Reply

Reply

p3

p1 p2 p1

p2 p3SECTION 15.2 DISTRIBUTED MUTUAL EXCLUSION 655

immediately. When receives ’s request, it finds that its own request has the lower

timestamp and so does not reply, holding off. However, finds that ’s request

has a lower timestamp than that of its own request and so replies immediately. On

receiving this second reply, can enter the critical section. When exits the critical

section, it will reply to ’s request and so grant it entry.

Gaining entry takes messages in this algorithm: to multicast the

request, followed by replies. Or, if there is hardware support for multicast, only

one message is required for the request; the total is then N messages. It is thus a more

expensive algorithm, in terms of bandwidth consumption, than the algorithms just

described. However, the client delay in requesting entry is again a round-trip time

(ignoring any delay incurred in multicasting the request message).

The advantage of this algorithm is that its synchronization delay is only one

message transmission time. Both the previous algorithms incurred a round-trip

synchronization delay.

The performance of the algorithm can be improved. First, note that the process

that last entered the critical section and that has received no other requests for it still goes

through the protocol as described, even though it could simply decide locally to reenter

the critical section. Second, Ricart and Agrawala refined this protocol so that it requires

N messages to obtain entry in the worst (and common) case, without hardware support

for multicast. This is described in Raynal [1988].

Maekawa’s voting algorithm • Maekawa [1985] observed that in order for a process to

enter a critical section, it is not necessary for all of its peers to grant it access. Processes

need only obtain permission to enter from subsets of their peers, as long as the subsets

used by any two processes overlap. We can think of processes as voting for one another

to enter the critical section. A ‘candidate’ process must collect sufficient votes to enter.

Processes in the intersection of two sets of voters ensure the safety property ME1, that

at most one process can enter the critical section, by casting their votes for only one

candidate.

Maekawa associated a voting set with each process ( ),

where . The sets are chosen so that, for all :

• •

– there is at least one common member of any two voting sets

• – to be fair, each process has a voting set of the same size

• Each process is contained in M of the voting sets .

Maekawa showed that the optimal solution, which minimizes K and allows the

processes to achieve mutual exclusion, has and M = K (so that each process is

in as many of the voting sets as there are elements in each one of those sets). It is nontrivial to calculate the optimal sets . As an approximation, a simple way of deriving

sets such that is to place the processes in a by matrix and let

be the union of the row and column containing .

Maekawa’s algorithm is shown in Figure 15.6. To obtain entry to the critical

section, a process sends request messages to all K members of (including itself).

cannot enter the critical section until it has received all K reply messages. When a

process in receives ’s request message, it sends a reply message immediately,

p2 p1

p1 p1 p2

p2 p2

p1

2( ) N – 1 N – 1

N – 1

Vi

pi i = 1 2 , , , … N

Vi

⊆ { } p1, , , p1 … pN Vi i j , = 1 2 , , , … N

pi ∈ Vi

Vi

Vj

∩ ≠ ∅

Vi

= K

pj Vi

K N ∼

R

i

R

i Ri ∼ 2 N N N

Vi

pi

pi Vi

pi

pj Vi pi656 CHAPTER 15 COORDINATION AND AGREEMENT

unless either its state is HELD or it has already replied (‘voted’) since it last received a

release message. Otherwise, it queues the request message (in the order of its arrival)

but does not yet reply. When a process receives a release message, it removes the head

of its queue of outstanding requests (if the queue is nonempty) and sends a reply

message (a ‘vote’) in response to it. To leave the critical section, sends release

messages to all K members of (including itself).

This algorithm achieves the safety property, ME1. If it were possible for two

processes and to enter the critical section at the same time, then the processes in

would have to have voted for both. But the algorithm allows a process to

make at most one vote between successive receipts of a release message, so this

situation is impossible.

Unfortunately, the algorithm is deadlock-prone. Consider three processes, ,

and , with , and . If the three

pi

Vi

Figure 15.6 Maekawa’s algorithm

On initialization

state := RELEASED;

voted := FALSE;

For to enter the critical section

state := WANTED;

Multicast request to all processes in ;

Wait until (number of replies received = K);

state := HELD;

On receipt of a request from at

if (state = HELD or voted = TRUE)

then

queue request from without replying;

else

send reply to ;

voted := TRUE;

end if

For to exit the critical section

state := RELEASED;

Multicast release to all processes in ;

On receipt of a release from at

if (queue of requests is non-empty)

then

remove head of queue – from , say;

send reply to ;

voted := TRUE;

else

voted := FALSE;

end if

pi

Vi

pi pj

pi

pi

pi

Vi

pi pj

pk

pk

pi pj

Vi

Vj

∩ ≠ ∅

p1 p2

p3 V1 = { } p1, p2 V2 = { } p2, p3 V3 = { } p3, p1SECTION 15.3 ELECTIONS 657

processes concurrently request entry to the critical section, then it is it is possible for

to reply to itself and hold off , for to reply to itself and hold off , and for to

reply to itself and hold off . Each process has received one out of two replies, and

none can proceed.

The algorithm can be adapted [Sanders 1987] so that it becomes deadlock-free. In

the adapted protocol, processes queue outstanding requests in happened-before order, so

that requirement ME3 is also satisfied.

The algorithm’s bandwidth utilization is messages per entry to the critical

section and messages per exit (assuming no hardware multicast facilities). The total

of is superior to the messages required by Ricart and Agrawala’s

algorithm, if N > 4. The client delay is the same as that of Ricart and Agrawala’s

algorithm, but the synchronization delay is worse: a round-trip time instead of a single

message transmission time.

Fault tolerance • The main points to consider when evaluating the above algorithms

with respect to fault tolerance are:

• What happens when messages are lost?

• What happens when a process crashes?

None of the algorithms that we have described would tolerate the loss of messages, if

the channels were unreliable. The ring-based algorithm cannot tolerate a crash failure of

any single process. As it stands, Maekawa’s algorithm can tolerate some process crash

failures: if a crashed process is not in a voting set that is required, then its failure will not

affect the other processes. The central server algorithm can tolerate the crash failure of

a client process that neither holds nor has requested the token. The Ricart and Agrawala

algorithm as we have described it can be adapted to tolerate the crash failure of such a

process, by taking it to grant all requests implicitly.

We invite the reader to consider how to adapt the algorithms to tolerate failures,

on the assumption that a reliable failure detector is available. Even with a reliable failure

detector, care is required to allow for failures at any point (including during a recovery

procedure), and to reconstruct the state of the processes after a failure has been detected.

For example, in the central-server algorithm, if the server fails it must be established

whether it or one of the client processes held the token.

We examine the general problem of how processes should coordinate their actions

in the presence of faults in Section 15.5.

15.3 Elections

An algorithm for choosing a unique process to play a particular role is called an election

algorithm. For example, in a variant of our central-server algorithm for mutual

exclusion, the ‘server’ is chosen from among the processes that

need to use the critical section. An election algorithm is needed for this choice. It is

essential that all the processes agree on the choice. Afterwards, if the process that plays

the role of server wishes to retire then another election is required to choose a

replacement.

p1

p2 p2 p3 p3

p1

2 N

N

3 N 2( ) N – 1

pi, ( ) i = 1 2 , , , … N658 CHAPTER 15 COORDINATION AND AGREEMENT

We say that a process calls the election if it takes an action that initiates a

particular run of the election algorithm. An individual process does not call more than

one election at a time, but in principle the N processes could call N concurrent elections.

At any point in time, a process is either a participant – meaning that it is engaged in

some run of the election algorithm – or a non-participant – meaning that it is not

currently engaged in any election.

An important requirement is for the choice of elected process to be unique, even

if several processes call elections concurrently. For example, two processes could

decide independently that a coordinator process has failed, and both call elections.

Without loss of generality, we require that the elected process be chosen as the one

with the largest identifier. The ‘identifier’ may be any useful value, as long as the

identifiers are unique and totally ordered. For example, we could elect the process with

the lowest computational load by having each process use < , i > as its identifier,

where load > 0 and the process index i is used to order identifiers with the same load.

Each process ( ) has a variable , which will contain the

identifier of the elected process. When the process first becomes a participant in an

election it sets this variable to the special value ‘ ’ to denote that it is not yet defined.

Our requirements are that, during any particular run of the algorithm:

E1: (safety) A participant process has or = P,

where P is chosen as the non-crashed process at the end of

the run with the largest identifier.

E2: (liveness) All processes participate and eventually either set

– or crash.

Note that there may be processes that are not yet participants, which record in

the identifier of the previous elected process.

We measure the performance of an election algorithm by its total network

bandwidth utilization (which is proportional to the total number of messages sent), and

by the turnaround time for the algorithm: the number of serialized message transmission

times between the initiation and termination of a single run.

A ring-based election algorithm • The algorithm of Chang and Roberts [1979] is

suitable for a collection of processes arranged in a logical ring. Each process has a

communication channel to the next process in the ring, , and all messages

are sent clockwise around the ring. We assume that no failures occur, and that the system

is asynchronous. The goal of this algorithm is to elect a single process called the

coordinator, which is the process with the largest identifier.

Initially, every process is marked as a non-participant in an election. Any process

can begin an election. It proceeds by marking itself as a participant, placing its identifier

in an election message and sending it to its clockwise neighbour.

When a process receives an election message, it compares the identifier in the

message with its own. If the arrived identifier is greater, then it forwards the message to

its neighbour. If the arrived identifier is smaller and the receiver is not a participant, then

it substitutes its own identifier in the message and forwards it; but it does not forward

the message if it is already a participant. On forwarding an election message in any case,

the process marks itself as a participant.

pi

1 ⁄ load

pi i = 1 2 , , , … N electedi

⊥ p

i electedi = ⊥ electedi

pi

elected

i ≠ ⊥

pj

elected

j

pi

p( ) i + 1 mod NSECTION 15.3 ELECTIONS 659

If, however, the received identifier is that of the receiver itself, then this process’s

identifier must be the greatest, and it becomes the coordinator. The coordinator marks

itself as a non-participant once more and sends an elected message to its neighbour,

announcing its election and enclosing its identity.

When a process receives an elected message, it marks itself as a nonparticipant, sets its variable to the identifier in the message and, unless it is the

new coordinator, forwards the message to its neighbour.

It is easy to see that condition E1 is met. All identifiers are compared, since a

process must receive its own identifier back before sending an elected message. For any

two processes, the one with the larger identifier will not pass on the other’s identifier. It

is therefore impossible that both should receive their own identifier back.

Condition E2 follows immediately from the guaranteed traversals of the ring

(there are no failures). Note how the non-participant and participant states are used so

that duplicate messages arising when two processes start an election at the same time are

extinguished as soon as possible, and always before the ‘winning’ election result has

been announced.

If only a single process starts an election, then the worst-performing case is when

its anti-clockwise neighbour has the highest identifier. A total of messages are

then required to reach this neighbour, which will not announce its election until its

identifier has completed another circuit, taking a further N messages. The elected

message is then sent N times, making messages in all. The turnaround time is

also , since these messages are sent sequentially.

Figure 15.7 A ring-based election in progress

Note: The election was started by process 17. The highest process identifier encountered

so far is 24. Participant processes are shown in a darker tint.

24

15

9

4

3

28

17

24

1

pi

elected

i

N – 1

3N – 1

3N – 1660 CHAPTER 15 COORDINATION AND AGREEMENT

An example of a ring-based election in progress is shown in Figure 15.7. The

election message currently contains 24, but process 28 will replace this with its identifier

when the message reaches it.

While the ring-based algorithm is useful for understanding the properties of

election algorithms in general, the fact that it tolerates no failures makes it of limited

practical value. However, with a reliable failure detector it is in principle possible to reconstitute the ring when a process crashes.

The bully algorithm • The bully algorithm [Garcia-Molina 1982] allows processes to

crash during an election, although it assumes that message delivery between processes

is reliable. Unlike the ring-based algorithm, this algorithm assumes that the system is

synchronous: it uses timeouts to detect a process failure. Another difference is that the

ring-based algorithm assumed that processes have minimal a priori knowledge of one

another: each knows only how to communicate with its neighbour, and none knows the

identifiers of the other processes. The bully algorithm, on the other hand, assumes that

each process knows which processes have higher identifiers, and that it can

communicate with all such processes.

There are three types of message in this algorithm: an election message is sent to

announce an election; an answer message is sent in response to an election message and

a coordinator message is sent to announce the identity of the elected process – the new

Figure 15.8 The bully algorithm

p1

p 2 p3 p4

p1

p

2 p3 p4

C

coordinator

Stage 4

C

election

election

Stage 2

p1

p

2 p3 p4

C

election

answer

answer

election

Stage 1

timeout

Stage 3

Eventually.....

p1

p

2 p3 p4

The election of coordinator p2, after the failure of p4 and then p3

election

answerSECTION 15.3 ELECTIONS 661

‘coordinator’. A process begins an election when it notices, through timeouts, that the

coordinator has failed. Several processes may discover this concurrently.

Since the system is synchronous, we can construct a reliable failure detector.

There is a maximum message transmission delay, , and a maximum delay for

processing a message . Therefore, we can calculate a time

that is an upper bound on the time that can elapse between

sending a message to another process and receiving a response. If no response arrives

within time T, then the local failure detector can report that the intended recipient of the

request has failed.

The process that knows it has the highest identifier can elect itself as the

coordinator simply by sending a coordinator message to all processes with lower

identifiers. On the other hand, a process with a lower identifier can begin an election by

sending an election message to those processes that have a higher identifier and awaiting

answer messages in response. If none arrives within time T, the process considers itself

the coordinator and sends a coordinator message to all processes with lower identifiers

announcing this. Otherwise, the process waits a further period for a coordinator

message to arrive from the new coordinator. If none arrives, it begins another election.

If a process receives a coordinator message, it sets its variable to the

identifier of the coordinator contained within it and treats that process as the coordinator.

If a process receives an election message, it sends back an answer message and

begins another election – unless it has begun one already.

When a process is started to replace a crashed process, it begins an election. If it

has the highest process identifier, then it will decide that it is the coordinator and

announce this to the other processes. Thus it will become the coordinator, even though

the current coordinator is functioning. It is for this reason that the algorithm is called the

‘bully’ algorithm.

The operation of the algorithm is shown in Figure 15.8. There are four processes,

– . Process detects the failure of the coordinator and announces an election

(stage 1 in the figure). On receiving an election message from , processes and

send answer messages to and begin their own elections; sends an answer

message to , but receives no answer message from the failed process (stage

2). It therefore decides that it is the coordinator. But before it can send out the

coordinator message, it too fails (stage 3). When ’s timeout period expires (which

we assume occurs before ’s timeout expires), it deduces the absence of a coordinator

message and begins another election. Eventually, is elected coordinator (stage 4).

This algorithm clearly meets the liveness condition E2, by the assumption of

reliable message delivery. And if no process is replaced, then the algorithm meets

condition E1. It is impossible for two processes to decide that they are the coordinator,

since the process with the lower identifier will discover that the other exists and defer to

it.

But the algorithm is not guaranteed to meet the safety condition E1 if processes

that have crashed are replaced by processes with the same identifiers. A process that

replaces a crashed process p may decide that it has the highest identifier just as another

process (which has detected p’s crash) decides that it has the highest identifier. Two

processes will therefore announce themselves as the coordinator concurrently.

Unfortunately, there are no guarantees on message delivery order, and the recipients of

these messages may reach different conclusions on which is the coordinator process.

T

trans

T

process

T 2T

= trans + Tprocess

T′

pi electedi

p1 p4 p1 p4

p1 p2 p3

p1 p3

p2 p3 p4

p1 T′

p2

p2662 CHAPTER 15 COORDINATION AND AGREEMENT

Furthermore, condition E1 may be broken if the assumed timeout values turn out

to be inaccurate – that is, if the processes’ failure detector is unreliable.

Taking the example just given, suppose that either had not failed but was just

running unusually slowly (that is, that the assumption that the system is synchronous is

incorrect), or that had failed but was then replaced. Just as sends its coordinator

message, (or its replacement) does the same. receives ‘s coordinator message

after it has sent its own and so sets . Due to variable message

transmission delays, receives ’s coordinator message after ’s and so

eventually sets . Condition E1 has been broken.

With regard to the performance of the algorithm, in the best case the process with

the second-highest identifier notices the coordinator’s failure. Then it can immediately

elect itself and send coordinator messages. The turnaround time is one message.

The bully algorithm requires messages in the worst case – that is, when the

process with the lowest identifier first detects the coordinator’s failure. For then

processes altogether begin elections, each sending messages to processes with higher

identifiers.

15.4 Coordination and agreement in group communication

This chapter examines the key coordination and agreement problems related to group

communication – that is, how to achieve the desired reliability and ordering properties

across all members of a group. Chapter 6 introduced group communication as an

example of an indirect communication technique whereby processes can send messages

to a group. This message is propagated to all members of the group with certain

guarantees in terms of reliability and ordering. We are particularly seeking reliability in

terms of the properties of validity, integrity and agreement, and ordering in terms of

FIFO ordering, causal ordering and total ordering.

In this chapter, we study multicast communication to groups of processes whose

membership is known. Chapter 18 will expand our study to fully fledged group

communication, including the management of dynamically varying groups.

System model • The system under consideration contains a collection of processes,

which can communicate reliably over one-to-one channels. As before, processes may

fail only by crashing.

The processes are members of groups, which are the destinations of messages sent

with the multicast operation. It is generally useful to allow processes to be members of

several groups simultaneously – for example, to enable processes to receive information

from several sources by joining several groups. But to simplify our discussion of

ordering properties, we shall sometimes restrict processes to being members of at most

one group at a time.

The operation multicast(g, m) sends the message m to all members of the group g

of processes. Correspondingly, there is an operation deliver(m) that delivers a message

sent by multicast to the calling process. We use the term deliver rather than receive to

make clear that a multicast message is not always handed to the application layer inside

p3

p3 p2

p3 p2 p3

elected

2 = p3

p1 p2 p3

elected

1 = p2

N – 2

O N ( ) 2

N – 1SECTION 15.4 COORDINATION AND AGREEMENT IN GROUP COMMUNICATION 663

the process as soon as it is received at the process’s node. This is explained when we

discuss multicast delivery semantics shortly.

Every message m carries the unique identifier of the process sender(m) that sent

it, and the unique destination group identifier group(m). We assume that processes do

not lie about the origin or destinations of messages.

Some algorithms assume that groups are closed (as defined in Chapter 6).

15.4.1 Basic multicast

It is useful to have at our disposal a basic multicast primitive that guarantees, unlike IP

multicast, that a correct process will eventually deliver the message, as long as the

multicaster does not crash. We call the primitive B-multicast and its corresponding basic

delivery primitive B-deliver. We allow processes to belong to several groups, and each

message is destined for some particular group.

A straightforward way to implement B-multicast is to use a reliable one-to-one

send operation, as follows:

To B-multicast(g, m): for each process , send(p, m);

On receive(m) at p: B-deliver(m) at p.

The implementation may use threads to perform the send operations concurrently, in an

attempt to reduce the total time taken to deliver the message. Unfortunately, such an

implementation is liable to suffer from a so-called ack-implosion if the number of

processes is large. The acknowledgements sent as part of the reliable send operation are

liable to arrive from many processes at about the same time. The multicasting process’s

buffers will rapidly fill, and it is liable to drop acknowledgements. It will therefore

retransmit the message, leading to yet more acknowledgements and further waste of

network bandwidth. A more practical basic multicast service can be built using IP

multicast, and we invite the reader to show this in Exercise 15.10.

15.4.2 Reliable multicast

Chapter 6 discussed reliable multicast in terms of validity, integrity and agreement. This

section builds on this informal discussion, presenting a more complete definition.

Following Hadzilacos and Toueg [1994] and Chandra and Toueg [1996], we

define a reliable multicast with corresponding operations R-multicast and R-deliver.

Properties analogous to integrity and validity are clearly highly desirable in reliable

multicast delivery, but we add another: a requirement that all correct processes in the

group must receive a message if any of them does. It is important to realize that this is

not a property of the B-multicast algorithm that is based on a reliable one-to-one send

operation. The sender may fail at any point while B-multicast proceeds, so some

processes may deliver a message while others do not.

A reliable multicast is one that satisfies the following properties:

Integrity: A correct process p delivers a message m at most once. Furthermore,

and m was supplied to a multicast operation by sender(m). (As with

one-to-one communication, messages can always be distinguished by a sequence

number relative to their sender.)

p ∈ g

p ∈ group m ( )664 CHAPTER 15 COORDINATION AND AGREEMENT

Validity: If a correct process multicasts message m, then it will eventually deliver m.

Agreement: If a correct process delivers message m, then all other correct processes

in group(m) will eventually deliver m.

The integrity property is analogous to that for reliable one-to-one communication. The

validity property guarantees liveness for the sender. This may seem an unusual property,

because it is asymmetric (it mentions only one particular process). But notice that

validity and agreement together amount to an overall liveness requirement: if one

process (the sender) eventually delivers a message m, since the correct processes agree

on the set of messages they deliver, it follows that m will eventually be delivered to all

the group’s correct members.

The advantage of expressing the validity condition in terms of self-delivery is

simplicity. What we require is that the message be delivered eventually by some correct

member of the group.

The agreement condition is related to atomicity, the property of ‘all or nothing’,

applied to delivery of messages to a group. If a process that multicasts a message crashes

before it has delivered it, then it is possible that the message will not be delivered to any

process in the group; but if it is delivered to some correct process, then all other correct

processes will deliver it. Many papers in the literature use the term ‘atomic’ to include

a total ordering condition; we define this shortly.

Implementing reliable multicast over B-multicast • Figure 15.9 gives a reliable multicast algorithm, with primitives R-multicast and R-deliver, that allows processes to belong to several closed groups simultaneously. To R-multicast a message, a process Bmulticasts the message to the processes in the destination group (including itself). When

the message is B-delivered, the recipient in turn B-multicasts the message to the group

(if it is not the original sender), and then R-delivers the message. Since a message may

arrive more than once, duplicates of the message are detected and not delivered.

This algorithm clearly satisfies the validity property, since a correct process will

eventually B-deliver the message to itself. By the integrity property of the underlying

communication channels used in B-multicast, the algorithm also satisfies the integrity

property.

Figure 15.9 Reliable multicast algorithm

On initialization

Received := {};

For process p to R-multicast message m to group g

B-multicast(g, m); // is included as a destination

On B-deliver(m) at process q with g = group(m)

if ( )

then

Received := ;

if ( ) then B-multicast(g, m); end if

R-deliver m;

end if

p ∈ g

m Received ∉

Received m ∪ { }

q p ≠SECTION 15.4 COORDINATION AND AGREEMENT IN GROUP COMMUNICATION 665

Agreement follows from the fact that every correct process B-multicasts the

message to the other processes after it has B-delivered it. If a correct process does not Rdeliver the message, then this can only be because it never B-delivered it. That in turn

can only be because no other correct process B-delivered it either; therefore none will

R-deliver it.

The reliable multicast algorithm that we have described is correct in an

asynchronous system, since we made no timing assumptions. But the algorithm is

inefficient for practical purposes. Each message is sent times to each process.

Reliable multicast over IP multicast • An alternative realization of R-multicast is to use

a combination of IP multicast, piggybacked acknowledgements (that is, acknowledgements attached to other messages) and negative acknowledgements. This R-multicast

protocol is based on the observation that IP multicast communication is often successful.

In the protocol, processes do not send separate acknowledgement messages; instead,

they piggyback acknowledgements on the messages that they send to the group. Processes send a separate response message only when they detect that they have missed a

message. A response indicating the absence of an expected message is known as a negative acknowledgement.

The description assumes that groups are closed. Each process p maintains a

sequence number for each group g to which it belongs. The sequence number is

initially zero. Each process also records , the sequence number of the latest message

it has delivered from process q that was sent to group g.

For p to R-multicast a message to group g, it piggybacks onto the message the

value and acknowledgements, of the form <q, >. An acknowledgement states, for

some sender q, the sequence number of the latest message from q destined for g that p

has delivered since it last multicast a message. The multicaster p then IP-multicasts the

message with its piggybacked values to g, and increments by one.

The piggybacked values in a multicast message enable the recipients to learn

about messages that they have not received. A process R-delivers a message destined for

g bearing the sequence number S from p if and only if , and it increments

by one immediately after delivery. If an arriving message has , then r has

delivered the message before and it discards it. If , or if for an

enclosed acknowledgement <q, R>, then there are one or more messages that it has not

yet received (and which are likely to have been dropped, in the first case). It keeps any

message for which in a hold-back queue (Figure 15.10) – such queues are

often used to meet message delivery guarantees. It requests missing messages by

sending negative acknowledgements, either to the original sender or to a process q from

which it has received an acknowledgement <q, > with no less than the required

sequence number.

The hold-back queue is not strictly necessary for reliability, but it simplifies the

protocol by enabling us to use sequence numbers to represent sets of delivered

messages. It also provides us with a guarantee of delivery order (see Section 15.4.3).

The integrity property follows from the detection of duplicates and the underlying

properties of IP multicast (which uses checksums to expunge corrupted messages). The

validity property holds because IP multicast has that property. For agreement we

require, first, that a process can always detect missing messages. That in turn means that

it will always receive a further message that enables it to detect the omission. As this

g

S

pg

R

qg

S

pg

R

qg

S

pg

S R

pg

= + 1

R

pg

S R

pg

≤

S R

pg

> + 1 R R

qg

>

S R

pg

> + 1

R

qg

R

qg666 CHAPTER 15 COORDINATION AND AGREEMENT

simplified protocol stands, we guarantee detection of missing messages only in the case

where correct processes multicast messages indefinitely. Second, the agreement

property requires that there is always an available copy of any message needed by a

process that did not receive it. We therefore assume that processes retain copies of the

messages they have delivered – indefinitely, in this simplified protocol.

Neither of the assumptions we made to ensure agreement is practical (see Exercise

15.15). However, agreement is practically addressed in the protocols from which ours is

derived: the Psync protocol [Peterson et al. 1989], Trans protocol [Melliar-Smith et al.

1990] and scalable reliable multicast protocol [Floyd et al. 1997]. Psync and Trans also

provide further delivery ordering guarantees.

Uniform properties • The definition of agreement given above refers only to the

behaviour of correct processes – processes that never fail. Consider what would happen

in the algorithm of Figure 15.9 if a process was not correct and crashed after it had Rdelivered a message. Since any process that R-delivers the message must first Bmulticast it, it follows that all correct processes will still eventually deliver the message.

Any property that holds whether or not processes are correct is called a uniform

property. We define uniform agreement as follows:

Uniform agreement: If a process, whether it is correct or fails, delivers message m,

then all correct processes in group(m) will eventually deliver m.

Uniform agreement allows a process to crash after it has delivered a message, while still

ensuring that all correct processes will deliver the message. We have argued that the

algorithm of Figure 15.9 satisfies this property, which is stronger than the non-uniform

agreement property defined above.

Uniform agreement is useful in applications where a process may take an action

that produces an observable inconsistency before it crashes. For example, suppose that

the processes are servers that manage copies of a bank account, and that updates to the

account are sent using reliable multicast to the group of servers. If the multicast does not

satisfy uniform agreement, then a client that accesses a server just before it crashes may

observe an update that no other server will process.

Figure 15.10 The hold-back queue for arriving multicast messages

Message

processing

Delivery queue

Hold-back

queue

deliver

Incoming

messages

When delivery

guarantees are

metSECTION 15.4 COORDINATION AND AGREEMENT IN GROUP COMMUNICATION 667

It is interesting to note that if we reverse the lines ‘R-deliver m’ and ‘if ( )

then B-multicast(g, m); end if’ in Figure 15.9, then the resultant algorithm does not

satisfy uniform agreement.

Just as there is a uniform version of agreement, there are also uniform versions of

any multicast property, including validity and integrity and the ordering properties that

we are about to define.

15.4.3 Ordered multicast

The basic multicast algorithm of Section 15.4.1 delivers messages to processes in an

arbitrary order, due to arbitrary delays in the underlying one-to-one send operations.

This lack of an ordering guarantee is not satisfactory for many applications. For

example, in a nuclear power plant it may be important that events signifying threats to

safety conditions and events signifying actions by control units are observed in the same

order by all processes in the system.

As discussed in Chapter 6, the common ordering requirements are total ordering,

causal ordering and FIFO ordering, together with hybrid solutions (in particular, totalcausal and total-FIFO). To simplify our discussion, we define these orderings under the

assumption that any process belongs to at most one group (later we discuss the

implications of allowing groups to overlap):

FIFO ordering: If a correct process issues multicast(g, m) and then multicast(g, ),

then every correct process that delivers will deliver m before .

Causal ordering: If multicast(g, m) multicast(g, ), where is the

happened-before relation induced only by messages sent between the members of g,

then any correct process that delivers will deliver m before .

Total ordering: If a correct process delivers message m before it delivers , then

any other correct process that delivers will deliver m before .

Causal ordering implies FIFO ordering, since any two multicasts by the same process

are related by happened-before. Note that FIFO ordering and causal ordering are only

partial orderings: not all messages are sent by the same process, in general; similarly,

some multicasts are concurrent (not ordered by happened-before).

Figure 15.11 illustrates the orderings for the case of three processes. Close

inspection of the figure shows that the totally ordered messages are delivered in the

opposite order to the physical time at which they were sent. In fact, the definition of total

ordering allows message delivery to be ordered arbitrarily, as long as the order is the

same at different processes. Since total ordering is not necessarily also a FIFO or causal

ordering, we define the hybrid of FIFO-total ordering as one for which message delivery

obeys both FIFO and total ordering; similarly, under causal-total ordering message

delivery obeys both causal and total ordering.

The definitions of ordered multicast do not assume or imply reliability. For

example, the reader should check that, under total ordering, if correct process p delivers

message m and then delivers , then a correct process q can deliver m without also

delivering or any other message ordered after m.

We can also form hybrids of ordered and reliable protocols. A reliable totally

ordered multicast is often referred to in the literature as an atomic multicast. Similarly,

q p ≠

m′

m′ m′

→ m′ →

m′ m′

m′

m′ m′

m′

m′668 CHAPTER 15 COORDINATION AND AGREEMENT

we may form reliable FIFO multicast, reliable causal multicast and reliable versions of

the hybrid ordered multicasts.

Ordering the delivery of multicast messages, as we shall see, can be expensive in

terms of delivery latency and bandwidth consumption. The ordering semantics that we

have described may delay the delivery of messages unnecessarily. That is, at the

application level, a message may be delayed for another message that it does not in fact

depend upon. For this reason, some have proposed multicast systems that use the

application-specific message semantics alone to determine the order of message

delivery [Cheriton and Skeen 1993, Pedone and Schiper 1999].

The example of the bulletin board • To make multicast delivery semantics more

concrete, consider an application in which users post messages to bulletin boards. Each

user runs a bulletin-board application process. Every topic of discussion has its own

process group. When a user posts a message to a bulletin board, the application

Figure 15.11 Total, FIFO and causal ordering of multicast messages

Notice the consistent ordering of totally ordered messages T1 and T2, the FIFO-related messages

F1 and F2 and the causally related messages C1 and C3 – and the otherwise arbitrary delivery

ordering of messages

F3

F1 F2

T2

T1

P1 P2 P3

Time

C3

C1

C2SECTION 15.4 COORDINATION AND AGREEMENT IN GROUP COMMUNICATION 669

multicasts the user’s posting to the corresponding group. Each user’s process is a

member of the group for the topic in which that user is interested, so they will receive

just the postings concerning that topic.

Reliable multicast is required if every user is to receive every posting eventually.

The users also have ordering requirements. Figure 15.12 shows the postings as they

appear to a particular user. At a minimum, FIFO ordering is desirable, since then every

posting from a given user – ‘A.Hanlon’, say – will be received in the same order, and

users can talk consistently about A.Hanlon’s second posting.

Note that the messages whose subjects are ‘Re: Microkernels’ (25) and ‘Re:

Mach’ (27) appear after the messages to which they refer. A causally ordered multicast

is needed to guarantee this relationship. Otherwise, arbitrary message delays could mean

that, say, the message ‘Re: Mach’ could appear before the original message about Mach.

If the multicast delivery was totally ordered, then the numbering in the lefthand

column would be consistent between users. Users could refer unambiguously, for

example, to ‘message 24’.

In practice, the USENET bulletin board system implements neither causal nor

total ordering. The communication costs of achieving these orderings on a large scale

outweigh their advantages.

Implementing FIFO ordering • FIFO-ordered multicast (with operations FO-multicast

and FO-deliver) is achieved with sequence numbers, much as we would achieve it for

one-to-one communication. We shall consider only non-overlapping groups. The reader

should verify that the reliable multicast protocol that we defined on top of IP multicast

in Section 15.4.2 also guarantees FIFO ordering, but we shall show how to construct a

FIFO-ordered multicast on top of any given basic multicast. We use the variables

and held at process p from the reliable multicast protocol of Section 15.4.2: is a

count of how many messages p has sent to g and, for each q, is the sequence number

of the latest message p has delivered from process q that was sent to group g.

For p to FO-multicast a message to group g, it piggybacks the value onto the

message, B-multicasts the message to g and then increments by 1. Upon receipt of a

message from q bearing the sequence number S, p checks whether . If so,

this message is the next one expected from the sender q and p FO-delivers it, setting

Figure 15.12 Display from bulletin board program

Bulletin board: os.interesting

Item From Subject

23 A.Hanlon Mach

24 G.Joseph Microkernels

25 A.Hanlon Re: Microkernels

26 T.L’Heureux RPC performance

27 M.Walker Re: Mach

end

S

pg

R

qg

pSg

R

qg

S

pg

S

pg

S R

qg

= + 1670 CHAPTER 15 COORDINATION AND AGREEMENT

:=S. If , it places the message in the hold-back queue until the intervening

messages have been delivered and .

Since all messages from a given sender are delivered in the same sequence, and

since a message’s delivery is delayed until its sequence number has been reached, the

condition for FIFO ordering is clearly satisfied. But this is so only under the assumption

that groups are non-overlapping.

Note that we can use any implementation of B-multicast in this protocol.

Moreover, if we use a reliable R-multicast primitive instead of B-multicast, then we

obtain a reliable FIFO multicast.

Implementing total ordering • The basic approach to implementing total ordering is to

assign totally ordered identifiers to multicast messages so that each process makes the

same ordering decision based upon these identifiers. The delivery algorithm is very

similar to the one we described for FIFO ordering; the difference is that processes keep

group-specific sequence numbers rather than process-specific sequence numbers. We

only consider how to totally order messages sent to non-overlapping groups. We call the

multicast operations TO-multicast and TO-deliver.

We discuss two main methods for assigning identifiers to messages. The first of

these is for a process called a sequencer to assign them (Figure 15.13). A process

wishing to TO-multicast a message m to group g attaches a unique identifier id(m) to it.

The messages for g are sent to the sequencer for g, sequencer(g), as well as to the

members of g. (The sequencer may be chosen to be a member of g.) The process

sequencer(g) maintains a group-specific sequence number , which it uses to assign

increasing and consecutive sequence numbers to the messages that it B-delivers.

It announces the sequence numbers by B-multicasting order messages to g (see Figure

15.13 for the details).

A message will remain in the hold-back queue indefinitely until it can be TOdelivered according to the corresponding sequence number. Since the sequence numbers

are well defined (by the sequencer), the criterion for total ordering is met. Furthermore,

if the processes use a FIFO-ordered variant of B-multicast, then the totally ordered

multicast is also causally ordered. We leave the reader to show this.

The obvious problem with a sequencer-based scheme is that the sequencer may

become a bottleneck and is a critical point of failure. Practical algorithms exist that

address the problem of failure. Chang and Maxemchuk [1984] first suggested a

multicast protocol employing a sequencer (which they called a token site). Kaashoek et

al. [1989] developed a sequencer-based protocol for the Amoeba system. These

protocols ensure that a message is in the hold-back queue at nodes before it is

delivered; up to f failures can thus be tolerated. Like Chang and Maxemchuk, Birman et

al. [1991] also employ a token-holding site that acts as a sequencer. The token can be

passed from process to process so that, for example, if only one process sends totally

ordered multicasts that process can act as the sequencer, saving communication.

The protocol of Kaashoek et al. uses hardware-based multicast – available on an

Ethernet, for example – rather than reliable point-to-point communication. In the

simplest variant of their protocol, processes send the message to be multicast to the

sequencer, one-to-one. The sequencer multicasts the message itself, as well as the

identifier and sequence number. This has the advantage that the other members of the

R

qg

S R

qg

> + 1

S R

qg

= + 1

s

g

f + 1SECTION 15.4 COORDINATION AND AGREEMENT IN GROUP COMMUNICATION 671

group receive only one message per multicast; its disadvantage is increased bandwidth

utilization. The protocol is described in full at www.cdk5.net/coordination.

The second method that we examine for achieving totally ordered multicast is one

in which the processes collectively agree on the assignment of sequence numbers to

messages in a distributed fashion. A simple algorithm – similar to one that was originally

developed to implement totally ordered multicast delivery for the ISIS toolkit [Birman

and Joseph 1987a] – is shown in Figure 15.14. Once more, a process B-multicasts its

message to the members of the group. The group may be open or closed. The receiving

processes propose sequence numbers for messages as they arrive and return these to the

sender, which uses them to generate agreed sequence numbers.

Each process q in group g keeps , the largest agreed sequence number it has

observed so far for group g, and , its own largest proposed sequence number. The

algorithm for process p to multicast a message m to group g is as follows:

1. p B-multicasts <m, i> to g, where i is a unique identifier for m.

2. Each process q replies to the sender p with a proposal for the message’s agreed

sequence number of := Max( , ) + 1. In reality, we must include process

identifiers in the proposed values to ensure a total order, since otherwise

different processes could propose the same integer value; but for the sake of

simplicity we shall not make that explicit here. Each process provisionally assigns

the proposed sequence number to the message and places it in its hold-back queue,

which is ordered with the smallest sequence number at the front.

Figure 15.13 Total ordering using a sequencer

1. Algorithm for group member p

On initialization: := 0;

To TO-multicast message m to group g

B-multicast( , <m, i>);

On B-deliver(<m, i>) with g = group(m)

Place <m, i> in hold-back queue;

On B-deliver( = <“order”, i, S>) with g = group( )

wait until <m, i> in hold-back queue and ;

TO-deliver m; // (after deleting it from the hold-back queue)

:= ;

2. Algorithm for sequencer of g

On initialization: := 0;

On B-deliver(<m, i>) with g = group(m)

B-multicast(g, <“order”, i, >);

:= ;

r

g

g sequencer g ∪ { } ( )

morder morder

S r

= g

r

g S + 1

s

g

s

g

s

g sg + 1

A

qg

P

qg

P

qg

A

qg

P

qg

P

qg672 CHAPTER 15 COORDINATION AND AGREEMENT

3. p collects all the proposed sequence numbers and selects the largest one, a, as the

next agreed sequence number. It then B-multicasts <i, a> to g. Each process q in

g sets := Max( , a) and attaches a to the message (which is identified by i).

It reorders the message in the hold-back queue if the agreed sequence number

differs from the proposed one. When the message at the front of the hold-back

queue has been assigned its agreed sequence number, it is transferred to the tail of

the delivery queue. Messages that have been assigned their agreed sequence

number but are not at the head of the hold-back queue are not yet transferred,

however.

If every process agrees the same set of sequence numbers and delivers them in the

corresponding order, then total ordering is satisfied. It is clear that correct processes

ultimately agree on the same set of sequence numbers, but we must show that they are

monotonically increasing and that no correct process can deliver a message prematurely.

Assume that a message has been assigned an agreed sequence number and has

reached the front of the hold-back queue. By construction, a message that is received

after this stage will and should be delivered after : it will have a larger proposed

sequence number and thus a larger agreed sequence number than . So let be any

other message that has not yet been assigned its agreed sequence number but that is on

the same queue. We have that:

agreedSequence( )  proposedSequence( )

by the algorithm just given. Since is at the front of the queue:

proposedSequence( ) > agreedSequence( )

Therefore:

agreedSequence( ) > agreedSequence( )

Figure 15.14 The ISIS algorithm for total ordering

FE

RM1

1

1

2

2

1

1

2

2

3

3

1 Message

2 Proposed Seq

P2

P3

P1

P4

3 Agreed Seq

A

qg

A

qg

m

1

m

1

m

1 m2

m

2 m2

m

1

m

2 m1

m

2 m1SECTION 15.4 COORDINATION AND AGREEMENT IN GROUP COMMUNICATION 673

and total ordering is assured.

This algorithm has higher latency than the sequencer-based multicast algorithm:

three messages are sent serially between the sender and the group before a message can

be delivered.

Note that the total ordering chosen by this algorithm is not also guaranteed to be

causally or FIFO-ordered: any two messages are delivered in an essentially arbitrary

total order, influenced by communication delays.

For other approaches to implementing total ordering, see Melliar-Smith et al.

[1990], Garcia-Molina and Spauster [1991] and Hadzilacos and Toueg [1994].

Implementing causal ordering • Next we give an algorithm for non-overlapping closed

groups based on that developed by Birman et al. [1991], shown in Figure 15.15, in which

the causally ordered multicast operations are CO-multicast and CO-deliver. The

algorithm takes account of the happened-before relationship only as it is established by

multicast messages. If the processes send one-to-one messages to one another, then

these will not be accounted for.

Each process ( ) maintains its own vector timestamp (see

Section 14.4). The entries in the timestamp count the number of multicast messages

from each process that happened-before the next message to be multicast.

To CO-multicast a message to group g, the process adds 1 to its entry in the

timestamp and B-multicasts the message along with its timestamp to g.

When a process B-delivers a message from , it must place it in the hold-back

queue before it can CO-deliver it – that is, until it is assured that it has delivered any

messages that causally preceded it. To establish this, waits until (a) it has delivered

any earlier message sent by , and (b) it has delivered any message that had

delivered at the time it multicast the message. Both of those conditions can be detected

by examining vector timestamps, as shown in Figure 15.15. Note that a process can

immediately CO-deliver to itself any message that it CO-multicasts, although this is not

described in Figure 15.15.

Figure 15.15 Causal ordering using vector timestamps

Algorithm for group member ( )

On initialization

:= 0 ( );

To CO-multicast message m to group g

:= ;

B-multicast(g, < , m>);

On B-deliver(< , m>) from ( ), with g = group(m)

place < , m> in hold-back queue;

wait until and ( );

CO-deliver m; // after removing it from the hold-back queue

:= ;

pi i = 1 2 , , … N

gVi

[ ] j j = 1 2 , , … N

gVi

[ ] i Vi g[ ] i + 1

gVi

gVj

pj j ≠ i

gVj

gVj

[ ] j = Vi g[ ] j + 1 Vj g[ ] k ≤ Vi g[ ] k k j ≠

gVi

[ ] j Vi g[ ] j + 1

pi i = 1 2 , , , … N

pi pj

pi

pj pj674 CHAPTER 15 COORDINATION AND AGREEMENT

Each process updates its vector timestamp upon delivering any message, to

maintain the count of causally precedent messages. It does this by incrementing the jth

entry in its timestamp by one. This is an optimization of the merge operation that appears

in the rules for updating vector clocks in Section 14.4. We can make the optimization in

view of the delivery condition in the algorithm of Figure 15.15, which guarantees that

only the jth entry will increase.

We outline the proof of the correctness of this algorithm as follows. Suppose that

multicast(g, m) multicast(g, ). Let V and be the vector timestamps of m and

, respectively. It is straightforward to prove inductively from the algorithm that

. In particular, if process multicast m, then .

Consider what happens when some correct process B-delivers (as opposed

to CO-delivering it) without first CO-delivering m. By the algorithm, can increase

only when delivers a message from , when it increases by 1. But has not

received m, and therefore cannot increase beyond . It is therefore not

possible for to CO-deliver , since this would require that , and

therefore that .

The reader should check that if we substitute the reliable R-multicast primitive in

place of B-multicast, then we obtain a multicast that is both reliable and causally

ordered.

Furthermore, if we combine the protocol for causal multicast with the sequencerbased protocol for totally ordered delivery, then we obtain message delivery that is both

total and causal. The sequencer delivers messages according to the causal order and

multicasts the sequence numbers for the messages in the order in which it receives them.

The processes in the destination group do not deliver a message until they have received

an order message from the sequencer and the message is next in the delivery sequence.

Since the sequencer delivers messages in causal order, and since all other

processes deliver messages in the same order as the sequencer, the ordering is indeed

both total and causal.

Overlapping groups • We have considered only non-overlapping groups in the

preceding definitions and algorithms for FIFO, total and causal ordering semantics. This

simplifies the problem, but it is not satisfactory, since in general processes need to be

members of multiple overlapping groups. For example, a process may be interested in

events from multiple sources and thus join a corresponding set of event-distribution

groups.

We can extend the ordering definitions to global orders [Hadzilacos and Toueg

1994], in which we have to consider that if message m is multicast to g, and if message

is multicast to , then both messages are addressed to the members of :

Global FIFO ordering: If a correct process issues multicast(g, m) and then

multicast( , ), then every correct process in that delivers will deliver

m before .

Global causal ordering: If multicast(g, m) multicast( , ), where is

the happened-before relation induced by any chain of multicast messages, then any

correct process in that delivers will deliver m before .

→ m′ V′

m′

V V < ′ pk V k [ ] ≤ V′[ ] k

pi m′

Vi

[ ] k

pi pk pi

Vi

[ ] k V k [ ] – 1

pi m′ Vi[ ] k ≥ V′[ ] k

Vi

[ ] k ≥ V k [ ]

m′ g′ g g ∩ ′

g′ m′ g g ∩ ′ m′

m′

→ g′ m′ →

g g ∩ ′ m′ m′SECTION 15.5 CONSENSUS AND RELATED PROBLEMS 675

Pairwise total ordering: If a correct process delivers message m sent to g before it

delivers sent to , then any other correct process in that delivers will

deliver m before .

Global total ordering: Let ‘<’ be the relation of ordering between delivery events.

We require that ‘<’ obeys pairwise total ordering and that it is acyclic – under

pairwise total ordering, ‘<’ is not acyclic by default.

One way of implementing these orders would be to multicast each message m to the

group of all processes in the system. Each process either discards or delivers the

message according to whether it belongs to group(m). This would be an inefficient and

unsatisfactory implementation: a multicast should involve as few processes as possible

beyond the members of the destination group. Alternatives are explored in Birman et al.

[1991], Garcia-Molina and Spauster [1991], Hadzilacos and Toueg [1994], Kindberg

[1995] and Rodrigues et al. [1998].

Multicast in synchronous and asynchronous systems • In this section, we have described

algorithms for reliable unordered multicast, (reliable) FIFO-ordered multicast, (reliable)

causally ordered multicast and totally ordered multicast. We also indicated how to

achieve a multicast that is both totally and causally ordered. We leave the reader to

devise an algorithm for a multicast primitive that guarantees both FIFO and total

ordering. All the algorithms that we have described work correctly in asynchronous

systems.

We did not, however, give an algorithm that guarantees both reliable and totally

ordered delivery. Surprising though it may seem, while possible in a synchronous

system, a protocol with these guarantees is impossible in an asynchronous distributed

system – even one that has at worst suffered a single process crash failure. We return to

this point in the next section.

15.5 Consensus and related problems

This section introduces the problem of consensus [Pease et al. 1980, Lamport et al.

1982] and the related problems of Byzantine generals and interactive consistency. We

refer to these collectively as problems of agreement. Roughly speaking, the problem is

for processes to agree on a value after one or more of the processes has proposed what

that value should be.

For example, in Chapter 2 we described a situation in which two armies should

decide consistently to attack the common enemy or retreat. Similarly, we may require

that all the correct processes controlling a spaceship’s engines should decide to either

‘proceed’ or ‘abort’ after each has proposed one action or the other, and in a transaction

to transfer funds from one account to another the processes involved must consistently

agree to perform the respective debit and credit. In mutual exclusion, the processes agree

on which process can enter the critical section. In an election, the processes agree on

which is the elected process. In totally ordered multicast, the processes agree on the

order of message delivery.

Protocols exist that are tailored to these individual types of agreement. We

described some of them above, and Chapters 16 and 17 examine transactions. But it is

m′ g′ g g ∩ ′ m′

m′676 CHAPTER 15 COORDINATION AND AGREEMENT

useful for us to consider more general forms of agreement, in a search for common

characteristics and solutions.

This section defines consensus more precisely and relates it to three related

agreement problems: Byzantine generals, interactive consistency and totally ordered

multicast. We go on to examine under what circumstances the problems can be solved,

and to sketch some solutions. In particular, we discuss the well-known impossibility

result of Fischer et al. [1985], which states that in an asynchronous system a collection

of processes containing only one faulty process cannot be guaranteed to reach

consensus. Finally, we consider how it is that practical algorithms exist despite the

impossibility result.

15.5.1 System model and problem definitions

Our system model includes a collection of processes ( ) communicating by message passing. An important requirement that applies in many practical situations is for consensus to be reached even in the presence of faults. We assume, as before,

that communication is reliable but that processes may fail. In this section we consider

Byzantine (arbitrary) process failures, as well as crash failures. We sometimes specify

an assumption that up to some number f of the N processes are faulty – that is, they exhibit some specified types of fault; the remainder of the processes are correct.

If arbitrary failures can occur, then another factor in specifying our system is

whether the processes digitally sign the messages that they send (see Section 11.4). If

processes sign their messages, then a faulty process is limited in the harm it can do.

Specifically, during an agreement algorithm it cannot make a false claim about the

values that a correct process has sent to it. The relevance of message signing will

become clearer when we discuss solutions to the Byzantine generals problem. By

default, we assume that signing does not take place.

Definition of the consensus problem • To reach consensus, every process begins in

the undecided state and proposes a single value , drawn from a set D

( ). The processes communicate with one another, exchanging values.

Each process then sets the value of a decision variable, . In doing so it enters the

decided state, in which it may no longer change ( ). Figure 15.16

shows three processes engaged in a consensus algorithm. Two processes propose

‘proceed’ and a third proposes ‘abort’ but then crashes. The two processes that remain

correct each decide ‘proceed’.

The requirements of a consensus algorithm are that the following conditions

should hold for every execution of it:

Termination: Eventually each correct process sets its decision variable.

Agreement: The decision value of all correct processes is the same: if and are

correct and have entered the decided state, then ( ).

Integrity: If the correct processes all proposed the same value, then any correct

process in the decided state has chosen that value.

Variations on the definition of integrity may be appropriate, according to the

application. For example, a weaker type of integrity would be for the decision value to

pi i = 1 2 , , , … N

pi

v

i

i = 1 2 , , , … N

d

i

d

i i = 1 2 , , , … N

pi pj

d

i = dj i j , = 1 2 , , , … NSECTION 15.5 CONSENSUS AND RELATED PROBLEMS 677

equal a value that some correct process proposed – not necessarily all of them. We use

the definition above except where stated otherwise. Integrity is also known as validity in

the literature.

To help in understanding how the formulation of the problem translates into an

algorithm, consider a system in which processes cannot fail. It is then straightforward to

solve consensus. For example, we can collect the processes into a group and have each

process reliably multicast its proposed value to the members of the group. Each process

waits until it has collected all N values (including its own). It then evaluates the function

, which returns the value that occurs most often among its

arguments, or the special value if no majority exists. Termination is guaranteed

by the reliability of the multicast operation. Agreement and integrity are guaranteed by

the definition of majority and the integrity property of a reliable multicast. Every process

receives the same set of proposed values, and every process evaluates the same function

of those values. So they must all agree, and if every process proposed the same value,

then they all decide on this value.

Note that majority is only one possible function that the processes could use to

agree upon a value from the candidate values. For example, if the values are ordered,

then the functions minimum and maximum may be appropriate.

If processes can crash this introduces the complication of detecting failures, and it

is not immediately clear that a run of the consensus algorithm can terminate. In fact, if

the system is asynchronous, then it may not; we shall return to this point shortly.

If processes can fail in arbitrary (Byzantine) ways, then faulty processes can in

principle communicate random values to the others. This may seem unlikely in practice,

but it is not beyond the bounds of possibility for a process with a bug to fail in this way.

Moreover, the fault may not be accidental, but the result of mischievous or malevolent

operation. Someone could deliberately make a process send different values to different

peers in an attempt to thwart the others, which are trying to reach consensus. In case of

inconsistency, correct processes must compare what they have received with what other

processes claim to have received.

Figure 15.16 Consensus for three processes

P2

P3 (crashes)

P1

Consensus algorithm

v1=proceed

v3=abort

v2=proceed

d1:=proceed d2:=proceed

majority v ( ) 1, , , v2 … vN

⊥ ∉ D678 CHAPTER 15 COORDINATION AND AGREEMENT

The Byzantine generals problem • In the informal statement of the Byzantine generals

problem [Lamport et al. 1982], three or more generals are to agree to attack or to retreat.

One, the commander, issues the order. The others, lieutenants to the commander, must

decide whether to attack or retreat. But one or more of the generals may be ‘treacherous’

– that is, faulty. If the commander is treacherous, he proposes attacking to one general

and retreating to another. If a lieutenant is treacherous, he tells one of his peers that the

commander told him to attack and another that they are to retreat.

The Byzantine generals problem differs from consensus in that a distinguished

process supplies a value that the others are to agree upon, instead of each of them

proposing a value. The requirements are:

Termination: Eventually each correct process sets its decision variable.

Agreement: The decision value of all correct processes is the same: if and are

correct and have entered the decided state, then ( ).

Integrity: If the commander is correct, then all correct processes decide on the value

that the commander proposed.

Note that, for the Byzantine generals problem, integrity implies agreement when the

commander is correct; but the commander need not be correct.

Interactive consistency • The interactive consistency problem is another variant of

consensus, in which every process proposes a single value. The goal of the algorithm is

for the correct processes to agree on a vector of values, one for each process. We call

this the ‘decision vector’. For example, the goal could be for each of a set of processes

to obtain the same information about their respective states.

The requirements for interactive consistency are:

Termination: Eventually each correct process sets its decision variable.

Agreement: The decision vector of all correct processes is the same.

Integrity: If is correct, then all correct processes decide on as the ith

component of their vector.

Relating consensus to other problems • Although it is common to consider the

Byzantine generals problem with arbitrary process failures, in fact each of the three

problems – consensus, Byzantine generals and interactive consistency – is meaningful

in the context of either arbitrary or crash failures. Similarly, each can be framed

assuming either a synchronous or an asynchronous system.

It is sometimes possible to derive a solution to one problem using a solution to

another. This is a very useful property, both because it increases our understanding of

the problems and because by reusing solutions we can potentially save on

implementation effort and complexity.

Suppose that there exist solutions to consensus (C), Byzantine generals (BG) and

interactive consistency (IC) as follows:

returns the decision value of in a run of the solution to the

consensus problem, where are the values that the processes proposed.

returns the decision value of in a run of the solution to the Byzantine

generals problem, where , the commander, proposes the value v.

pi pj

d

i = dj i j , = 1 2 , , , … N

pi vi

C

i( ) v1, , , v2 … vN pi

v

1, , , v2 … vN

BG

i( ) j v , pi

pjSECTION 15.5 CONSENSUS AND RELATED PROBLEMS 679

returns the jth value in the decision vector of in a run of the

solution to the interactive consistency problem, where are the values

that the processes proposed.

The definitions of , and assume that a faulty process proposes a single

notional value, even though it may have given different proposed values to each of the

other processes. This is only a convenience: the solutions will not rely on any such

notional value.

It is possible to construct solutions out of the solutions to other problems. We give

three examples:

IC from BG: We construct a solution to IC from BG by running BG N times, once

with each process ( ) acting as the commander:

( )

C from IC: For the case where a majority of processes are correct, we construct a

solution to C from IC by running IC to produce a vector of values at each process,

then applying an appropriate function on the vector’s values to derive a single value:

where and majority is as defined above.

BG from C: We construct a solution to BG from C as follows:

• The commander sends its proposed value v to itself and each of the

remaining processes.

• All processes run C with the values that they receive ( may be

faulty).

• They derive ( ).

The reader should check that the termination, agreement and integrity conditions are

preserved in each case. Fischer [1983] relates the three problems in more detail.

In systems with crash failures, consensus is equivalent to solving reliable and

totally ordered multicast: given a solution to one, we can solve the other. Implementing

consensus with a reliable and totally ordered multicast operation RTO-multicast is

straightforward. We collect all the processes into a group, g. To achieve consensus, each

process performs RTO-multicast(g, ). Then each process chooses ,

where is the first value that RTO-delivers. The termination property follows from

the reliability of the multicast. The agreement and integrity properties follow from the

reliability and total ordering of multicast delivery. Chandra and Toueg [1996]

demonstrate how reliable and totally ordered multicast can be derived from consensus.

15.5.2 Consensus in a synchronous system

This section describes an algorithm to solve consensus in a synchronous system,

although it is based on a modified form of the integrity requirement. The algorithm uses

only a basic multicast protocol. It assumes that up to f of the N processes exhibit crash

failures.

IC

i( ) v1, , , v2 … vN [ ] j pi

v

1, , , v2 … vN

C

i BGi ICi

pi i j , = 1 2 , , , … N

IC

i( ) v1, , , v2 … vN [ ] j = BGi( ) j v , j i j , = 1 2 , , , … N

C

i( ) v1, , … vN = majority IC ( ) i( ) v1, , … vN [ ] … 1 , , ICi( ) v1, , … vN [ ] N

i = 1 2 , …N

pj

v

1, , , v2 … vN pj

BG

i( ) j v , = Ci( ) v1, , , v2 … vN i = 1 2 , , , … N

pi vi pi di = mi

m

i pi680 CHAPTER 15 COORDINATION AND AGREEMENT

To reach consensus, each correct process collects proposed values from the other

processes. The algorithm proceeds in rounds, in each of which the correct

processes B-multicast the values between themselves. At most f processes may crash, by

assumption. At worst, all f crashes will occur during the rounds, but the algorithm

guarantees that at the end of the rounds all the correct processes that have survived will

be in a position to agree.

The algorithm, shown in Figure 15.17, is based on that by Dolev and Strong

[1983] and its presentation by Attiya and Welch [1998]. Their modified form of the

integrity requirement applies to the proposed values of all processes, not just the correct

ones: if all processes, whether correct or not, proposed the same value, then any correct

process in the decided state would chose that value. Given that the algorithm assumes

crash failures at worst, the proposed values of correct and non-correct processes would

not be expected to differ, at least not on the basis of failures. The revised form of

integrity enables the convenient use of the minimum function to choose a decision value

from those proposed.

The variable holds the set of proposed values known to process at the

beginning of round r. Each process multicasts the set of values that it has not sent in

previous rounds. It then takes delivery of similar multicast messages from other

processes and records any new values. Although this is not shown in Figure 15.17, the

duration of a round is limited by setting a timeout based on the maximum time for a

correct process to multicast a message. After rounds, each process chooses the

minimum value it has received as its decision value.

Termination is obvious from the fact that the system is synchronous. To check the

correctness of the algorithm, we must show that each process arrives at the same set of

values at the end of the final round. Agreement and integrity (in its modified form) will

then follow, because the processes apply the minimum function to this set.

f + 1

Figure 15.17 Consensus in a synchronous system

Algorithm for process ; algorithm proceeds in rounds

On initialization

:= ; = {};

In round r ( )

B-multicast(g, ); // Send only values that have not been sent

:= ;

while (in round r)

{

On B-deliver( ) from some

:= ;

}

After rounds

Assign ;

pi ∈ g f + 1

Values

1i

v

{ } i Valuesi 0

1 ≤ ≤ r f + 1

Values

ri

Values

ri

– 1

–

Values

ri

+ 1

Values

ri

Vj

pj

Values

ri

+ 1

Values

ri

+ 1

Vj

∪

( ) f + 1

d

i = minimum Values ( ) i f + 1

Valuesi r pi

f + 1SECTION 15.5 CONSENSUS AND RELATED PROBLEMS 681

Assume, to the contrary, that two processes differ in their final set of values.

Without loss of generality, some correct process possesses a value v that another

correct process ( ) does not possess. The only explanation for possessing a

proposed value v at the end that does not possess is that any third process, , say,

that managed to send v to crashed before v could be delivered to . In turn, any

process sending v in the previous round must have crashed, to explain why possesses

v in that round but did not receive it. Proceeding in this way, we have to posit at least

one crash in each of the preceding rounds. But we have assumed that at most f crashes

can occur, and there are rounds. We have arrived at a contradiction.

It turns out that any algorithm to reach consensus despite up to f crash failures

requires at least rounds of message exchanges, no matter how it is constructed

[Dolev and Strong 1983]. This lower bound also applies in the case of Byzantine failures

[Fischer and Lynch 1982].

15.5.3 The Byzantine generals problem in a synchronous system

Now we discuss the Byzantine generals problem in a synchronous system. Unlike the

algorithm for consensus described in the previous section, here we assume that

processes can exhibit arbitrary failures. That is, a faulty process may send any message

with any value at any time; and it may omit to send any message. Up to f of the N

processes may be faulty. Correct processes can detect the absence of a message through

a timeout; but they cannot conclude that the sender has crashed, since it may be silent

for some time and then send messages again.

We assume that the communication channels between pairs of processes are

private. If a process could examine all the messages that other processes sent, then it

could detect the inconsistencies in what a faulty process sends to different processes.

Our default assumption of channel reliability means that no faulty process can inject

messages into the communication channel between correct processes.

Lamport et al. [1982] considered the case of three processes that send unsigned

messages to one another. They showed that there is no solution that guarantees to meet

the conditions of the Byzantine generals problem if one process is allowed to fail. They

generalized this result to show that no solution exists if N  3f. We shall demonstrate

these results shortly. They went on to give an algorithm that solves the Byzantine

generals problem in a synchronous system if , for unsigned (they call them

‘oral’) messages.

Impossibility with three processes • Figure 15.18 shows two scenarios in which just one

of three processes is faulty. In the lefthand configuration one of the lieutenants, , is

faulty; on the right the commander, , is faulty. Each scenario in Figure 15.18 shows

two rounds of messages: the values the commander sends, and the values that the

lieutenants subsequently send to each other. The numeric prefixes serve to specify the

sources of messages and to show the different rounds. Read the ‘:’ symbol in messages

as ‘says’; for example, ‘3:1:u’ is the message ‘3 says 1 says u’.

In the lefthand scenario, the commander correctly sends the same value v to each

of the other two processes, and correctly echoes this to . However, sends a

value to . All knows at this stage is that it has received differing values; it

cannot tell which were sent out by the commander.

pi

pj i j ≠ pi

pj pk

pi pj

pk

pj

f + 1

f + 1

N ≥ 3f + 1

p3

p1

p2 p3 p3

u v ≠ p2 p2682 CHAPTER 15 COORDINATION AND AGREEMENT

In the righthand scenario, the commander is faulty and sends differing values to

the lieutenants. After has correctly echoed the value x that it received, is in the

same situation as it was in when was faulty: it has received two differing values.

If a solution exists, then process is bound to decide on value v when the

commander is correct, by the integrity condition. If we accept that no algorithm can

possibly distinguish between the two scenarios, must also choose the value sent by

the commander in the righthand scenario.

Following exactly the same reasoning for , assuming that it is correct, we are

forced to conclude (by symmetry) that also chooses the value sent by the commander

as its decision value. But this contradicts the agreement condition (the commander sends

differing values if it is faulty). So no solution is possible.

Note that this argument rests on our intuition that nothing can be done to improve

a correct general’s knowledge beyond the first stage, where it cannot tell which process

is faulty. It is possible to prove the correctness of this intuition [Pease et al. 1980].

Byzantine agreement can be reached for three generals, with one of them faulty, if the

generals digitally sign their messages.

Impossibility with N ≤ 3f • Pease et al. generalized the basic impossibility result for

three processes, to prove that no solution is possible if N  3f. In outline, the argument

is as follows. Assume that a solution exists with N  3f. Let each of three processes ,

and use the solution to simulate the behaviour of , and generals,

respectively, where and . Assume, furthermore, that

one of the three processes is faulty. Those of , and that are correct simulate

correct generals: they simulate the interactions of their own generals internally and send

messages from their generals to those simulated by other processes. The faulty process’s

simulated generals are faulty: the messages that it sends as part of the simulation to the

other two processes may be spurious. Since and , at most f

simulated generals are faulty.

Because the algorithm that the processes run is assumed to be correct, the

simulation terminates. The correct simulated generals (in the two correct processes)

agree and satisfy the integrity property. But now we have a means for the two correct

processes out of the three to reach consensus: each decides on the value chosen by all of

their simulated generals. This contradicts our impossibility result for three processes,

with one faulty.

Figure 15.18 Three Byzantine generals

p1 (Commander)

p2 p3

1:v 1:v

2:1:v

3:1:u

p1 (Commander)

p2 p3

1:w 1:x

2:1:w

3:1:x

Faulty processes are shown in grey

p3 p2

p3

p2

p2

p3

p3

p1

p2 p3 n1 n2 n3

n

1 + + n2 n3 = N n1, , n2 n3 ≤ N/3

p1 p2 p3

N ≤ 3f n1, , n2 n3 ≤ N/3SECTION 15.5 CONSENSUS AND RELATED PROBLEMS 683

Solution with one faulty process • There is not sufficient space to describe fully the

algorithm of Pease et al. that solves the Byzantine generals problem in a synchronous

system with . Instead, we give the operation of the algorithm for the case

, and illustrate it for , .

The correct generals reach agreement in two rounds of messages:

• In the first round, the commander sends a value to each of the lieutenants.

• In the second round, each of the lieutenants sends the value it received to its peers.

A lieutenant receives a value from the commander, plus values from its peers. If

the commander is faulty, then all the lieutenants are correct and each will have gathered

exactly the set of values that the commander sent out. Otherwise, one of the lieutenants

is faulty; each of its correct peers receives copies of the value that the commander

sent, plus a value that the faulty lieutenant sent to it.

In either case, the correct lieutenants need only apply a simple majority function

to the set of values they receive. Since N  4,  2. Therefore, the majority

function will ignore any value that a faulty lieutenant sent, and it will produce the value

that the commander sent if the commander is correct.

We now illustrate the algorithm that we have just outlined for the case of four

generals. Figure 15.19 shows two scenarios similar to those in Figure 15.18, but in this

case there are four processes, one of which is faulty. As in Figure 15.18, in the lefthand

configuration one of the lieutenants, , is faulty; on the right, the commander, , is

faulty.

In the lefthand case, the two correct lieutenant processes agree, deciding on the

commander's value:

decides on

decides on

In the righthand case the commander is faulty, but the three correct processes agree:

, and decide on (the special value applies

where no majority of values exists).

N ≥ 3f + 1

N ≥ 4 f = 1 N = 4 f = 1

N – 2

N – 2

( ) N – 2

Figure 15.19 Four Byzantine generals

p1 (Commander)

p2 p3

1:v 1:v

2:1:v

3:1:u

Faulty processes are shown in grey

p4

1:v

4:1:v

2:1:v 3:1:w

4:1:v

p1 (Commander)

p2 p3

1:u 1:w

2:1:u

3:1:w

p4

1:v

4:1:v

2:1:u 3:1:w

4:1:v

p3 p1

p2 majority v u v ( ) , , = v

p4 majority v v w ( ) , , = v

p2 p3 p4 majority u v w ( ) ⊥ , , = ⊥684 CHAPTER 15 COORDINATION AND AGREEMENT

The algorithm takes account of the fact that a faulty process may omit to send a message.

If a correct process does not receive a message within a suitable time limit (the system

is synchronous), it proceeds as though the faulty process had sent it the value .

Discussion • We can measure the efficiency of a solution to the Byzantine generals

problem – or any other agreement problem – by asking:

• How many message rounds does it take? (This is a factor in how long it takes for

the algorithm to terminate.)

• How many messages are sent, and of what size? (This measures the total

bandwidth utilization and has an impact on the execution time.)

In the general case ( ) the Lamport et al. [1982] algorithm for unsigned messages

operates over rounds. In each round, a process sends to a subset of the other

processes the values that it received in the previous round. The algorithm is very costly:

it involves sending messages.

Fischer and Lynch [1982] proved that any deterministic solution to consensus

assuming Byzantine failures (and hence to the Byzantine generals problem, as Section

15.5.1 showed) will take at least message rounds. So no algorithm can operate

faster in this respect than that of Lamport et al. But there have been improvements in the

message complexity, for example Garay and Moses [1993].

Several algorithms, such as that of Dolev and Strong [1983], take advantage of

signed messages. Dolev and Strong’s algorithm again takes rounds, but the

number of messages sent is only .

The complexity and cost of the solutions suggest that they are applicable only

where the threat is great. Solutions that are based on more detailed knowledge of the

fault model may be more efficient [Barborak et al. 1993]. If malicious users are the

source of the threat, then a system to counter them is likely to use digital signatures; a

solution without signatures is impractical.

15.5.4 Impossibility in asynchronous systems

We have provided solutions to consensus and the Byzantine generals problem (and

hence, by derivation, to interactive consistency). However, all these solutions relied

upon the system being synchronous. The algorithms assume that message exchanges

take place in rounds, and that processes are entitled to time out and assume that a faulty

process has not sent them a message within the round, because the maximum delay has

been exceeded.

Fischer et al. [1985] proved that no algorithm can guarantee to reach consensus in

an asynchronous system, even with one process crash failure. In an asynchronous

system, processes can respond to messages at arbitrary times, so a crashed process is

indistinguishable from a slow one. Their proof, which is beyond the scope of this book,

involves showing that there is always some continuation of the processes’ execution that

avoids consensus being reached.

We immediately know from the result of Fischer et al. that there is no guaranteed

solution in an asynchronous system to the Byzantine generals problem, to interactive

consistency or to totally ordered and reliable multicast. If there were such a solution

⊥

f ≥ 1

f + 1

O N ( ) f + 1

f + 1

f + 1

O N ( ) 2SECTION 15.5 CONSENSUS AND RELATED PROBLEMS 685

then, by the results of Section 15.5.1, we would have a solution to consensus –

contradicting the impossibility result.

Note the word ‘guarantee’ in the statement of the impossibility result. The result

does not mean that processes can never reach distributed consensus in an asynchronous

system if one is faulty. It allows that consensus can be reached with some probability

greater than zero, confirming what we know in practice. For example, despite the fact

that our systems are often effectively asynchronous, transaction systems have been

reaching consensus regularly for many years.

One approach to working around the impossibility result is to consider partially

synchronous systems, which are sufficiently weaker than synchronous systems to be

useful as models of practical systems, and sufficiently stronger than asynchronous

systems for consensus to be solvable in them [Dwork et al. 1988]. That approach is

beyond the scope of this book. However, we shall now outline three other techniques for

working around the impossibility result: fault masking, and reaching consensus by

exploiting failure detectors and by randomizing aspects of the processes’ behaviour.

Masking faults • The first technique is to avoid the impossibility result altogether by

masking any process failures that occur (see Section 2.4.2 for an introduction to fault

masking). For example, transaction systems employ persistent storage, which survives

crash failures. If a process crashes, then it is restarted (automatically, or by an

administrator). The process places sufficient information in persistent storage at critical

points in its program so that if it should crash and be restarted, it will find sufficient data

to be able to continue correctly with its interrupted task. In other words, it will behave

like a process that is correct, but that sometimes takes a long time to perform a

processing step.

Of course, fault masking is generally applicable in system design. Chapter 16

discusses how transactional systems take advantage of persistent storage. Chapter 18

describes how process failures can also be masked by replicating software components.

Consensus using failure detectors • Another method for circumventing the

impossibility result is to employ failure detectors. Some practical systems employ

‘perfect by design’ failure detectors to reach consensus. No failure detector in an

asynchronous system that works solely by message passing can really be perfect.

However, processes can agree to deem a process that has not responded for more than a

bounded time to have failed. An unresponsive process may not really have failed, but

the remaining processes act as if it had done. They make the failure ‘fail-silent’ by

discarding any subsequent messages that they do in fact receive from a ‘failed’ process.

In other words, we have effectively turned an asynchronous system into a synchronous

one. This technique is used in the ISIS system [Birman 1993].

This method relies upon the failure detector usually being accurate. When it is

inaccurate, then the system has to proceed without a group member that otherwise could

potentially have contributed to the system’s effectiveness. Unfortunately, making the

failure detector reasonably accurate involves using long timeout values, forcing

processes to wait a relatively long time (and not perform useful work) before concluding

that a process has failed. Another issue that arises for this approach is network

partitioning, which we discuss in Chapter 18.

A quite different approach is to use imperfect failure detectors, and to reach

consensus while allowing suspected processes to behave correctly instead of excluding686 CHAPTER 15 COORDINATION AND AGREEMENT

them. Chandra and Toueg [1996] analyzed the properties that a failure detector must

have in order to solve the consensus problem in an asynchronous system. They showed

that consensus can be solved in an asynchronous system, even with an unreliable failure

detector, if fewer than processes crash and communication is reliable. The weakest

type of failure detector for which this is so is called an eventually weak failure detector.

This is one that is both:

Eventually weakly complete: Each faulty process is eventually suspected

permanently by some correct process.

Eventually weakly accurate: After some point in time, at least one correct process is

never suspected by any correct process.

Chandra and Toueg show that we cannot implement an eventually weak failure detector

in an asynchronous system by message passing alone. However, we described a

message-based failure detector in Section 15.1 that adapts its timeout values according

to observed response times. If a process or the connection to it is very slow, then the

timeout value will grow so that cases of falsely suspecting a process become rare. In the

case of many real systems, this algorithm behaves sufficiently closely to an eventually

weak failure detector for practical purposes.

Chandra and Toueg’s consensus algorithm allows falsely suspected processes to

continue their normal operations and allows processes that have suspected them to

receive messages from them and process those messages normally. This makes the

application programmer’s life complicated, but it has the advantage that correct

processes are not wasted by being falsely excluded. Moreover, timeouts for detecting

failures can be set less conservatively than with the ISIS approach.

Consensus using randomization • The result of Fischer et al. [1985] depends on what

we can consider to be an ‘adversary’. This is a ‘character’ (actually, just a collection of

random events) who can exploit the phenomena of asynchronous systems so as to foil

the processes’ attempts to reach consensus. The adversary manipulates the network to

delay messages so that they arrive at just the wrong time, and similarly it slows down or

speeds up the processes just enough so that they are in the ‘wrong’ state when they

receive a message.

The third technique that addresses the impossibility result is to introduce an

element of chance in the processes’ behaviour, so that the adversary cannot exercise its

thwarting strategy effectively. Consensus might still not be reached in some cases, but

this method enables processes to reach consensus in a finite expected time. A

probabilistic algorithm that solves consensus even with Byzantine failures can be found

in Canetti and Rabin [1993].

N ⁄ 2SECTION 15.6 SUMMARY 687

15.6 Summary

We began this chapter by discussing the need for processes to access shared resources

under conditions of mutual exclusion. Locks are not always implemented by the servers

that manage the shared resources, and a separate distributed mutual exclusion service is

then required. Three algorithms were considered that achieve mutual exclusion: one

employing a central server, a ring-based algorithm and a multicast-based algorithm

using logical clocks. None of these mechanisms can withstand failure as we described

them, although they can be modified to tolerate some faults.

Next we explored elections, considering a ring-based algorithm and the bully

algorithm, whose common aim is to elect a process uniquely from a given set – even if

several elections take place concurrently. The bully algorithm could be used, for

example, to elect a new master time server, or a new lock server, when the previous one

fails.

The following section described coordination and agreement in group

communication. It discussed reliable multicast, in which the correct processes agree on

the set of messages to be delivered, and multicast with FIFO, causal and total delivery

ordering. We gave algorithms for reliable multicast and for all three types of delivery

ordering.

Finally, we described the three problems of consensus, Byzantine generals and

interactive consistency. We defined the conditions for their solution and we showed

relationships between these problems – including the relationship between consensus

and reliable, totally ordered multicast.

Solutions exist in a synchronous system, and we described some of them. In fact,

solutions exist even when arbitrary failures are possible.We outlined part of the solution

to the Byzantine generals problem of Lamport et al. [1982]. More recent algorithms

have lower complexity, but in principle none can better the rounds taken by this

algorithm, unless messages are digitally signed.

The chapter ended by describing the fundamental result of Fischer et al. [1982]

concerning the impossibility of guaranteeing consensus in an asynchronous system. We

discussed how it is that, nonetheless, systems regularly do reach agreement in

asynchronous systems.

EXERCISES

15.1 Give an example where an unreliable failure detector produces a suspected value when

the system is actually functioning correctly. page 648

15.2 For a system with N processes pi, I = 1, 2, 3 … N, what are the essential requirements

for the mutual exclusion conditions ME1 and ME2? page 648

15.3 Give a formula for the maximum throughput of a mutual exclusion system in terms of

the synchronization delay. page 651

15.4 In the central server algorithm for mutual exclusion, describe a situation in which two

requests are not processed in happened-before order. page 652

f + 1688 CHAPTER 15 COORDINATION AND AGREEMENT

15.5 Adapt the central server algorithm for mutual exclusion to handle the crash failure of any

client (in any state), assuming that the server is correct and given a reliable failure detector.

Comment on whether the resultant system is fault-tolerant. What would happen if a client

that possesses the token is wrongly suspected to have failed? page 652

15.6 Even without a deadlock, a poor algorithm might lead to starvation. Give an example of

a system which leads to starvation. page 650

15.7 In a certain distributed system, each process typically uses mutual exclusion to remove the

critical section problem. Different algorithms are available for mutual exclusion. Explain

how the central server algorithm is used to achieve mutual exclusion. How does an algorithm

using multicast and logical clocks differ from the central server algorithm? page 655

15.8 Maekawa’s voting algorithm does not need permission from all peer processes to gain

access. Processes need only obtain permission from subsets of their peers to enter, as

long as the subsets used by any two processes overlap. Is this algorithm deadlock-prone?

Give an example. pages 656, 657

15.9 Provide a mechanism to overcome the limitations of Maekawa’s voting algorithm.

page 657

15.10 Devise an algorithm to implement an unreliable failure detector. page 648

15.11 How is the fault tolerance of different mutual exclusion algorithms evaluated? Name the

type of failure that cannot be tolerated by the ring-based algorithm. page 657

15.12 Explain why reversing the order of the lines ‘R-deliver m’ and ‘if ( ) then Bmulticast(g, m); end if’ in Figure 15.9 makes the algorithm no longer satisfy uniform

agreement. Does the reliable multicast algorithm based on IP multicast satisfy uniform

agreement? page 664

15.13 Explain whether the algorithm for reliable multicast over IP multicast works for open as

well as closed groups. Given any algorithm for closed groups, how, simply, can we

derive an algorithm for open groups? page 665

15.14 Explain how to adapt the algorithm for reliable multicast over IP multicast to eliminate

the hold-back queue – so that a received message that is not a duplicate can be delivered

immediately, but without any ordering guarantees. Hint: use sets of sequence numbers

to represent the messages that have been delivered so far. page 665

15.15 Consider how to address the impractical assumptions we made in order to meet the

validity and agreement properties for the reliable multicast protocol based on IP

multicast. Hint: add a rule for deleting retained messages when they have been delivered

everywhere, and consider adding a dummy ‘heartbeat’ message, which is never

delivered to the application, but which the protocol sends if the application has no

message to send. page 665

15.16 Show that the FIFO-ordered multicast algorithm does not work for overlapping groups,

by considering two messages sent from the same source to two overlapping groups, and

considering a process in the intersection of those groups. Adapt the protocol to work for

this case. Hint: processes should include with their messages the latest sequence

numbers of messages sent to all groups. page 670

q p ≠EXERCISES 689

15.17 Show that, if the basic multicast that we use in the algorithm of Figure 15.13 is also

FIFO-ordered, then the resultant totally-ordered multicast is also causally ordered. Is it

the case that any multicast that is both FIFO-ordered and totally ordered is thereby

causally ordered? page 671

15.18 Suggest how to adapt the causally ordered multicast protocol to handle overlapping

groups. page 673

15.19 In discussing Maekawa’s mutual exclusion algorithm, we gave an example of three

subsets of a set of three processes that could lead to a deadlock. Use these subsets as

multicast groups to show how a pairwise total ordering is not necessarily acyclic.

page 674

15.20 Construct a solution to reliable, totally ordered multicast in a synchronous system, using

a reliable multicast and a solution to the consensus problem. page 675

15.21 A basic multicast primitive guarantees that a correct process will eventually deliver the

message, as long as the multicaster does not crash. Show the use of a reliable one-to-one

send operation to implement B-multicast. page 663

15.22 Consider the algorithm given in Figure 15.17 for consensus in a synchronous system,

which uses the following integrity definition: if all processes, whether correct or not,

proposed the same value, then any correct process in the decided state would chose that

value. Now consider an application in which correct processes may propose different

results, e.g., by running different algorithms to decide which action to take in a control

system’s operation. Suggest an appropriate modification to the integrity definition and

thus to the algorithm. page 680

15.23 Show that Byzantine agreement can be reached for three generals, with one of them

faulty, if the generals digitally sign their messages. page 681This page intentionally left blank691

16

TRANSACTIONS AND

CONCURRENCY CONTROL

16.1 Introduction

16.2 Transactions

16.3 Nested transactions

16.4 Locks

16.5 Optimistic concurrency control

16.6 Timestamp ordering

16.7 Comparison of methods for concurrency control

16.8 Summary

This chapter discusses the application of transactions and concurrency control to shared

objects managed by servers.

A transaction defines a sequence of server operations that is guaranteed by the

server to be atomic in the presence of multiple clients and server crashes. Nested

transactions are structured from sets of other transactions. They are particularly useful in

distributed systems because they allow additional concurrency.

All of the concurrency control protocols are based on the criterion of serial

equivalence and are derived from rules for conflicts between operations. Three methods

are described:

• Locks are used to order transactions that access the same objects according to the

order of arrival of their operations at the objects.

• Optimistic concurrency control allows transactions to proceed until they are ready

to commit, whereupon a check is made to see whether they have performed

conflicting operations on objects.

• Timestamp ordering uses timestamps to order transactions that access the same

objects according to their starting times.692 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

16.1 Introduction

The goal of transactions is to ensure that all of the objects managed by a server remain

in a consistent state when they are accessed by multiple transactions and in the presence

of server crashes. Chapter 2 introduced a failure model for distributed systems.

Transactions deal with crash failures of processes and omission failures in

communication, but not any type of arbitrary (or Byzantine) behaviour. The failure

model for transactions is presented in Section 16.1.2.

Objects that can be recovered after their server crashes are called recoverable

objects. In general, the objects managed by a server may be stored in volatile memory

(for example, RAM) or persistent memory (for example, a hard disk). Even if objects

are stored in volatile memory, the server may use persistent memory to store sufficient

information for the state of the objects to be recovered if the server process crashes. This

enables servers to make objects recoverable. A transaction is specified by a client as a

set of operations on objects to be performed as an indivisible unit by the servers

managing those objects. The servers must guarantee that either the entire transaction is

carried out and the results recorded in permanent storage or, in the case that one or more

of them crashes, its effects are completely erased. The next chapter discusses issues

related to transactions that involve several servers, in particular how they decide on the

outcome of a distributed transaction. This chapter concentrates on the issues for a

transaction at a single server. A client’s transaction is also regarded as indivisible from

the point of view of other clients’ transactions in the sense that the operations of one

transaction cannot observe the partial effects of the operations of another. Section 16.1.1

discusses simple synchronization of access to objects, and Section 16.2 introduces

transactions, which require more advanced techniques to prevent interference between

clients. Section 16.3 discusses nested transactions. Sections 16.4 to 16.6 discuss three

methods of concurrency control for transactions whose operations are all addressed to a

single server (locks, optimistic concurrency control and timestamp ordering). Chapter

17 discusses how these methods are extended for use with transactions whose operations

are addressed to several servers.

To explain some of the points made in this chapter, we use a banking example,

shown in Figure 16.1. Each account is represented by a remote object whose interface,

Account, provides operations for making deposits and withdrawals and for enquiring

about and setting the balance. Each branch of the bank is represented by a remote object

whose interface, Branch, provides operations for creating a new account, for looking up

an account by name and for enquiring about the total funds at that branch.

16.1.1 Simple synchronization (without transactions)

One of the main issues of this chapter is that unless a server is carefully designed, its

operations performed on behalf of different clients may sometimes interfere with one

another. Such interference may result in incorrect values in the objects. In this section,

we discuss how client operations may be synchronized without recourse to transactions.

Atomic operations at the server • We have seen in earlier chapters that the use of

multiple threads is beneficial to performance in many servers. We have also noted that

the use of threads allows operations from multiple clients to run concurrently andSECTION 16.1 INTRODUCTION 693

possibly access the same objects. Therefore, the methods of objects should be designed

for use in a multi-threaded context. For example, if the methods deposit and withdraw

are not designed for use in a multi-threaded program, then it is possible that the actions

of two or more concurrent executions of the method could be interleaved arbitrarily and

have strange effects on the instance variables of the account objects.

Chapter 7 explains the use of the synchronized keyword, which can be applied to

methods in Java to ensure that only one thread at a time can access an object. In our

example, the class that implements the Account interface will be able to declare the

methods as synchronized. For example:

public synchronized void deposit(int amount) throws RemoteException{

// adds amount to the balance of the account

}

If one thread invokes a synchronized method on an object, then that object is effectively

locked, and another thread that invokes one of its synchronized methods will be blocked

until the lock is released. This form of synchronization forces the execution of threads

to be separated in time and ensures that the instance variables of a single object are

accessed in a consistent manner. Without synchronization, two separate deposit

invocations might read the balance before either has incremented it – resulting in an

incorrect value. Any method that accesses an instance variable that can vary should be

synchronized.

Operations that are free from interference from concurrent operations being

performed in other threads are called atomic operations. The use of synchronized

Figure 16.1 Operations of the Account interface

deposit(amount)

deposit amount in the account

withdraw(amount)

withdraw amount from the account

getBalance()→ amount

return the balance of the account

setBalance(amount)

set the balance of the account to amount

Operations of the Branch interface

create(name)→ account

create a new account with a given name

lookUp(name)→ account

return a reference to the account with the given name

branchTotal()→ amount

return the total of all the balances at the branch694 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

methods in Java is one way of achieving atomic operations. But in other programming

environments for multi-threaded servers the operations on objects still need to have

atomic operations in order to keep their objects consistent. This may be achieved by the

use of any available mutual exclusion mechanism, such as a mutex.

Enhancing client cooperation by synchronization of server operations • Clients may use

a server as a means of sharing some resources. This is achieved by some clients using

operations to update the server’s objects and other clients using operations to access

them. The above scheme for synchronized access to objects provides all that is required

in many applications – it prevents threads interfering with one another. However, some

applications require a way for threads to communicate with each other.

For example, a situation may arise in which the operation requested by one client

cannot be completed until an operation requested by another client has been performed.

This can happen when some clients are producers and others are consumers – the

consumers may have to wait until a producer has supplied some more of the commodity

in question. It can also occur when clients are sharing a resource – clients needing the

resource may have to wait for other clients to release it. We shall see later in this chapter

that a similar situation arises when locks or timestamps are used for concurrency control

in transactions.

The Java wait and notify methods introduced in Chapter 7 allow threads to

communicate with one another in a manner that solves the above problems. They must

be used within synchronized methods of an object. A thread calls wait on an object so

as to suspend itself and to allow another thread to execute a method of that object. A

thread calls notify to inform any thread waiting on that object that it has changed some

of its data. Access to an object is still atomic when threads wait for one another: a thread

that calls wait gives up its lock and suspends itself as a single atomic action; when a

thread is restarted after being notified it acquires a new lock on the object and resumes

execution from after its wait. A thread that calls notify (from within a synchronized

method) completes the execution of that method before releasing the lock on the object.

Consider the implementation of a shared Queue object with two methods: first

removes and returns the first object in the queue, and append adds a given object to the

end of the queue. The method first will test whether the queue is empty, in which case

it will call wait on the queue. If a client invokes first when the queue is empty, it will not

get a reply until another client has added something to the queue – the append operation

will call notify when it has added an object to the queue. This allows one of the threads

waiting on the queue object to resume and to return the first object in the queue to its

client. When threads can synchronize their actions on an object by means of wait and

notify, the server holds onto requests that cannot immediately be satisfied and the client

waits for a reply until another client has produced whatever it needs.

In Section 16.4, we discuss the implementation of a lock as an object with

synchronized operations. When clients attempt to acquire a lock, they can be made to

wait until the lock is released by other clients.

Without the ability to synchronize threads in this way, a client that cannot be

satisfied immediately – for example, a client that invokes the first method on an empty

queue – is told to try again later. This is unsatisfactory, because it will involve the client

in polling the server and the server in carrying out extra requests. It is also potentially

unfair because other clients may make their requests before the waiting client tries again.SECTION 16.2 TRANSACTIONS 695

16.1.2 Failure model for transactions

Lampson [1981] proposed a fault model for distributed transactions that accounts for

failures of disks, servers and communication. In this model, the claim is that the

algorithms work correctly in the presence of predictable faults, but no claims are made

about their behaviour when a disaster occurs. Although errors may occur, they can be

detected and dealt with before any incorrect behaviour results. The model states the

following:

• Writes to permanent storage may fail, either by writing nothing or by writing a

wrong value – for example, writing to the wrong block is a disaster. File storage

may also decay. Reads from permanent storage can detect (by a checksum) when

a block of data is bad.

• Servers may crash occasionally. When a crashed server is replaced by a new

process, its volatile memory is first set to a state in which it knows none of the

values (for example, of objects) from before the crash. After that it carries out a

recovery procedure using information in permanent storage and obtained from

other processes to set the values of objects including those related to the two-phase

commit protocol (see Section 17.6). When a processor is faulty, it is made to crash

so that it is prevented from sending erroneous messages and from writing wrong

values to permanent storage – that is, so it cannot produce arbitrary failures.

Crashes can occur at any time; in particular, they may occur during recovery.

• There may be an arbitrary delay before a message arrives. A message may be lost,

duplicated or corrupted. The recipient can detect corrupted messages using a

checksum. Both forged messages and undetected corrupt messages are regarded

as disasters.

The fault model for permanent storage, processors and communications was used to

design a stable system whose components can survive any single fault and present a

simple failure model. In particular, stable storage provided an atomic write operation in

the presence of a single fault of the write operation or a crash failure of the process. This

was achieved by replicating each block on two disk blocks. A write operation was

applied to the pair of disk blocks, and in the case of a single fault, one good block was

always available. A stable processor used stable storage to enable it to recover its

objects after a crash. Communication errors were masked by using a reliable remote

procedure calling mechanism.

16.2 Transactions

In some situations, clients require a sequence of separate requests to a server to be

atomic in the sense that:

1. They are free from interference by operations being performed on behalf of other

concurrent clients.

2. Either all of the operations must be completed successfully or they must have no

effect at all in the presence of server crashes.696 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

We return to our banking example to illustrate transactions. A client that performs a

sequence of operations on a particular bank account on behalf of a user will first lookUp

the account by name and then apply the deposit, withdraw and getBalance operations

directly to the relevant account. In our examples, we use accounts with names A, B and

C. The client looks them up and stores references to them in variables a, b and c of type

Account. The details of looking up the accounts by name and the declarations of the

variables are omitted from the examples.

Figure 16.2 shows an example of a simple client transaction specifying a series of

related actions involving the bank accounts A, B and C. The first two actions transfer

$100 from A to B and the second two transfer $200 from C to B. A client achieves a

transfer operation by doing a withdrawal followed by a deposit.

Transactions originate from database management systems. In that context, a

transaction is an execution of a program that accesses a database. Transactions were

introduced to distributed systems in the form of transactional file servers such as XDFS

[Mitchell and Dion 1982]. In the context of a transactional file server, a transaction is an

execution of a sequence of client requests for file operations. Transactions on distributed

objects were provided in several research systems, including Argus [Liskov 1988] and

Arjuna [Shrivastava et al. 1991]. In this last context, a transaction consists of the

execution of a sequence of client requests such as, for example, those in Figure 16.2.

From the client’s point of view, a transaction is a sequence of operations that forms a

single step, transforming the server data from one consistent state to another.

Transactions can be provided as a part of middleware. For example, CORBA

provides the specification for an Object Transaction Service [OMG 2003] with IDL

interfaces allowing clients’ transactions to include multiple objects at multiple servers.

The client is provided with operations to specify the beginning and end of a transaction.

The client maintains a context for each transaction, which it propagates with each

operation in that transaction. In CORBA, transactional objects are invoked within the

scope of a transaction and generally have some persistent store associated with them.

In all of these contexts, a transaction applies to recoverable objects and is intended

to be atomic. It is often called an atomic transaction. There are two aspects to atomicity:

All or nothing: A transaction either completes successfully, in which case the effects

of all of its operations are recorded in the objects, or (if it fails or is deliberately

aborted) has no effect at all. This all-or-nothing effect has two further aspects of its

own:

Failure atomicity: The effects are atomic even when the server crashes.

Figure 16.2 A client’s banking transaction

Transaction T:

a.withdraw(100);

b.deposit(100);

c.withdraw(200);

b.deposit(200);SECTION 16.2 TRANSACTIONS 697

Durability: After a transaction has completed successfully, all its effects are

saved in permanent storage. We use the term ‘permanent storage’ to refer to files

held on disk or another permanent medium. Data saved in a file will survive if the

server process crashes.

Isolation: Each transaction must be performed without interference from other

transactions; in other words, the intermediate effects of a transaction must not be

visible to other transactions. The box below introduces a mnemonic, ACID, for

remembering the properties of atomic transactions.

To support the requirement for failure atomicity and durability, the objects must be

recoverable; that is, when a server process crashes unexpectedly due to a hardware fault

or a software error, the changes due to all completed transactions must be available in

permanent storage so that when the server is replaced by a new process, it can recover

the objects to reflect the all-or-nothing effect. By the time a server acknowledges the

completion of a client’s transaction, all of the transaction’s changes to the objects must

have been recorded in permanent storage.

A server that supports transactions must synchronize the operations sufficiently to

ensure that the isolation requirement is met. One way of doing this is to perform the

transactions serially – one at a time, in some arbitrary order. Unfortunately, this solution

would generally be unacceptable for servers whose resources are shared by multiple

interactive users. For instance, in our banking example it is desirable to allow several

bank clerks to perform online banking transactions at the same time as one another.

The aim for any server that supports transactions is to maximize concurrency.

Therefore transactions are allowed to execute concurrently if this would have the same

effect as a serial execution – that is, if they are serially equivalent or serializable.

ACID properties

Härder and Reuter [1983] suggested the mnemonic ‘ACID’ to remember the

properties of transactions, which are as follows:

Atomicity: a transaction must be all or nothing;

Consistency: a transaction takes the system from one consistent state to another

consistent state;

Isolation;

Durability.

We have not included ‘consistency’ in our list of the properties of transactions

because it is generally the responsibility of the programmers of servers and clients to

ensure that transactions leave the database consistent.

As an example of consistency, suppose that in the banking example, an object

holds the sum of all the account balances and its value is used as the result of

branchTotal. Clients can get the sum of all the account balances either by using

branchTotal or by calling getBalance on each of the accounts. For consistency, they

should get the same result from both methods. To maintain this consistency, the

deposit and withdraw operations must update the object holding the sum of all the

account balances.698 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

Transaction capabilities can be added to servers of recoverable objects. Each

transaction is created and managed by a coordinator, which implements the Coordinator

interface shown in Figure 16.3. The coordinator gives each transaction an identifier, or

TID. The client invokes the openTransaction method of the coordinator to introduce a

new transaction – a transaction identifier or TID is allocated and returned. At the end of

a transaction, the client invokes the closeTransaction method to indicate its end – all of

the recoverable objects accessed by the transaction should be saved. If, for some reason,

the client wants to abort a transaction, it invokes the abortTransaction method – all of

its effects should be removed from sight.

A transaction is achieved by cooperation between a client program, some

recoverable objects and a coordinator. The client specifies the sequence of invocations

on recoverable objects that are to comprise a transaction. To achieve this, the client

sends with each invocation the transaction identifier returned by openTransaction. One

way to make this possible is to include an extra argument in each operation of a

recoverable object to carry the TID. For example, in the banking service the deposit

operation might be defined:

deposit(trans, amount)

Deposits amount in the account for transaction with TID trans

When transactions are provided as middleware, the TID can be passed implicitly

with all remote invocations between openTransaction and closeTransaction or

abortTransaction. This is what the CORBA Transaction Service does. We shall not

show TIDs in our examples.

Normally, a transaction completes when the client makes a closeTransaction

request. If the transaction has progressed normally, the reply states that the transaction

is committed – this constitutes a promise to the client that all of the changes requested in

the transaction are permanently recorded and that any future transactions that access the

same data will see the results of all of the changes made during the transaction.

Alternatively, the transaction may have to abort for one of several reasons related

to the nature of the transaction itself, to conflicts with another transaction or to the

crashing of a process or computer. When a transaction is aborted the parties involved

(the recoverable objects and the coordinator) must ensure that none of its effects are

visible to future transactions, either in the objects or in their copies in permanent storage.

A transaction is either successful or is aborted in one of two ways – the client

aborts it (using an abortTransaction call to the server) or the server aborts it. Figure 16.4

Figure 16.3 Operations in the Coordinator interface

openTransaction() → trans;

Starts a new transaction and delivers a unique TID trans. This identifier will be used

in the other operations in the transaction.

closeTransaction(trans)→ (commit, abort);

Ends a transaction: a commit return value indicates that the transaction has

committed; an abort return value indicates that it has aborted.

abortTransaction(trans);

Aborts the transaction.SECTION 16.2 TRANSACTIONS 699

shows these three alternative life histories for transactions. We refer to a transaction as

failing in both of the latter cases.

Service actions related to process crashes • If a server process crashes unexpectedly, it

is eventually replaced. The new server process aborts any uncommitted transactions and

uses a recovery procedure to restore the values of the objects to the values produced by

the most recently committed transaction. To deal with a client that crashes unexpectedly

during a transaction, servers can give each transaction an expiry time and abort any

transaction that has not completed before its expiry time.

Client actions related to server process crashes • If a server crashes while a transaction

is in progress, the client will become aware of this when one of the operations returns an

exception after a timeout. If a server crashes and is then replaced during the progress of

a transaction, the transaction will no longer be valid and the client must be informed via

an exception to the next operation. In either case, the client must then formulate a plan,

possibly in consultation with the human user, for the completion or abandonment of the

task of which the transaction was a part.

16.2.1 Concurrency control

This section illustrates two well-known problems of concurrent transactions in the

context of the banking example – the ‘lost update’ problem and the ‘inconsistent

retrievals’ problem. We then show how both of these problems can be avoided by using

serially equivalent executions of transactions. We assume throughout that each of the

operations deposit, withdraw, getBalance and setBalance is a synchronized operation –

that is, that its effects on the instance variable that records the balance of an account are

atomic.

The lost update problem • The lost update problem is illustrated by the following pair

of transactions on bank accounts A, B and C, whose initial balances are $100, $200 and

$300, respectively. Transaction T transfers an amount from account A to account B.

Transaction U transfers an amount from account C to account B. In both cases, the

Figure 16.4 Transaction life histories

Successful Aborted by client Aborted by server

openTransaction openTransaction openTransaction

operation operation operation

operation operation operation

• • server aborts •

• • transaction → •

operation operation operation ERROR

reported to client

closeTransaction abortTransaction700 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

amount transferred is calculated to increase the balance of B by 10%. The net effects on

account B of executing the transactions T and U should be to increase the balance of

account B by 10% twice, so its final value is $242.

Now consider the effects of allowing the transactions T and U to run concurrently,

as in Figure 16.5. Both transactions get the balance of B as $200 and then deposit $20.

The result is incorrect, increasing the balance of account B by $20 instead of $42. This

is an illustration of the ‘lost update’ problem. U’s update is lost because T overwrites it

without seeing it. Both transactions have read the old value before either writes the new

value.

In Figure 16.5 onwards, we show the operations that affect the balance of an

account on successive lines down the page, and the reader should assume that an

operation on a particular line is executed at a later time than the one on the line above it.

Inconsistent retrievals • Figure 16.6 shows another example related to a bank account

in which transaction V transfers a sum from account A to B and transaction W invokes

the branchTotal method to obtain the sum of the balances of all the accounts in the bank.

Figure 16.5 The lost update problem

Transaction T: Transaction U:

balance = b.getBalance();

b.setBalance(balance\*1.1);

a.withdraw(balance/10)

balance = b.getBalance();

b.setBalance(balance\*1.1);

c.withdraw(balance/10)

balance = b.getBalance(); $200

balance = b.getBalance(); $200

b.setBalance(balance\*1.1); $220

b.setBalance(balance\*1.1); $220

a.withdraw(balance/10) $80

c.withdraw(balance/10) $280

Figure 16.6 The inconsistent retrievals problem

Transaction V: Transaction W:

a.withdraw(100)

b.deposit(100)

aBranch.branchTotal()

a.withdraw(100); $100

total = a.getBalance() $100

total = total + b.getBalance() $300

total = total + c.getBalance()

b.deposit(100) $300 •

•SECTION 16.2 TRANSACTIONS 701

The balances of the two bank accounts, A and B, are both initially $200. The result of

branchTotal includes the sum of A and B as $300, which is wrong. This is an illustration

of the ‘inconsistent retrievals’ problem. W’s retrievals are inconsistent because V has

performed only the withdrawal part of a transfer at the time the sum is calculated.

Serial equivalence • If each of several transactions is known to have the correct effect

when it is done on its own, then we can infer that if these transactions are done one at a

time in some order the combined effect will also be correct. An interleaving of the

operations of transactions in which the combined effect is the same as if the transactions

had been performed one at a time in some order is a serially equivalent interleaving.

When we say that two different transactions have the same effect as one another, we

mean that the read operations return the same values and that the instance variables of

the objects have the same values at the end.

The use of serial equivalence as a criterion for correct concurrent execution

prevents the occurrence of lost updates and inconsistent retrievals.

The lost update problem occurs when two transactions read the old value of a

variable and then use it to calculate the new value. This cannot happen if one transaction

is performed before the other, because the later transaction will read the value written

by the earlier one. As a serially equivalent interleaving of two transactions produces the

same effect as a serial one, we can solve the lost update problem by means of serial

equivalence. Figure 16.7 shows one such interleaving in which the operations that affect

the shared account, B, are actually serial, for transaction T does all its operations on B

before transaction U does. Another interleaving of T and U that has this property is one

in which transaction U completes its operations on account B before transaction T starts.

We now consider the effect of serial equivalence in relation to the inconsistent

retrievals problem, in which transaction V is transferring a sum from account A to B and

transaction W is obtaining the sum of all the balances (see Figure 16.6). The inconsistent

retrievals problem can occur when a retrieval transaction runs concurrently with an

update transaction. It cannot occur if the retrieval transaction is performed before or

after the update transaction. A serially equivalent interleaving of a retrieval transaction

and an update transaction, for example as in Figure 16.8, will prevent inconsistent

retrievals occurring.

Figure 16.7 A serially equivalent interleaving of T and U

Transaction T: Transaction U:

balance = b.getBalance()

b.setBalance(balance\*1.1)

a.withdraw(balance/10)

balance = b.getBalance()

b.setBalance(balance\*1.1)

c.withdraw(balance/10)

balance = b.getBalance() $200

b.setBalance(balance\*1.1) $220

balance = b.getBalance() $220

b.setBalance(balance\*1.1) $242

a.withdraw(balance/10) $80

c.withdraw(balance/10) $278702 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

Conflicting operations • When we say that a pair of operations conflicts we mean that

their combined effect depends on the order in which they are executed. To simplify

matters we consider a pair of operations, read and write. read accesses the value of an

object and write changes its value. The effect of an operation refers to the value of an

object set by a write operation and the result returned by a read operation. The conflict

rules for read and write operations are given in Figure 16.9.

For any pair of transactions, it is possible to determine the order of pairs of

conflicting operations on objects accessed by both of them. Serial equivalence can be

defined in terms of operation conflicts as follows:

For two transactions to be serially equivalent, it is necessary and sufficient that all

pairs of conflicting operations of the two transactions be executed in the same

order at all of the objects they both access.

Figure 16.8 A serially equivalent interleaving of V and W

Transaction V: Transaction W:

a.withdraw(100);

b.deposit(100)

aBranch.branchTotal( )

a.withdraw(100); $100

b.deposit(100) $300

total = a.getBalance() $100

total = total + b.getBalance() $400

total = total + c.getBalance()

...

Figure 16.9 Read and write operation conflict rules

Operations of different

transactions

Conflict Reason

read read No

Because the effect of a pair of read operations does

not depend on the order in which they are executed

read write Yes

Because the effect of a read and a write operation

depends on the order of their execution

write write Yes

Because the effect of a pair of write operations

depends on the order of their executionSECTION 16.2 TRANSACTIONS 703

Consider as an example the transactions T and U, defined as follows:

T: x = read(i); write(i, 10); write(j, 20);

U: y = read(j); write(j, 30); z = read (i);

Then consider the interleaving of their executions, shown in Figure 16.10. Note that

each transaction’s access to objects i and j is serialized with respect to one another,

because T makes all of its accesses to i before U does and U makes all of its accesses to

j before T does. But the ordering is not serially equivalent, because the pairs of

conflicting operations are not done in the same order at both objects. Serially equivalent

orderings require one of the following two conditions:

1. T accesses i before U and T accesses j before U.

2. U accesses i before T and U accesses j before T.

Serial equivalence is used as a criterion for the derivation of concurrency control

protocols. These protocols attempt to serialize transactions in their access to objects.

Three alternative approaches to concurrency control are commonly used: locking,

optimistic concurrency control and timestamp ordering. However, most practical

systems use locking, which is discussed in Section 16.4. When locking is used, the

server sets a lock, labelled with the transaction identifier, on each object just before it is

accessed and removes these locks when the transaction has completed. While an object

is locked, only the transaction that it is locked for can access that object; other

transactions must either wait until the object is unlocked or, in some cases, share the

lock. The use of locks can lead to deadlocks, with transactions waiting for each other to

release locks – for example, when a pair of transactions each has an object locked that

the other needs to access. We discuss the deadlock problem and some remedies for it in

Section 16.4.1.

Optimistic concurrency control is described in Section 16.5. In optimistic

schemes, a transaction proceeds until it asks to commit, and before it is allowed to

commit the server performs a check to discover whether it has performed operations on

any objects that conflict with the operations of other concurrent transactions, in which

case the server aborts it and the client may restart it. The aim of the check is to ensure

that all the objects are correct.

Timestamp ordering is described in Section 16.6. In timestamp ordering, a server

records the most recent time of reading and writing of each object and for each

Figure 16.10 A non–serially-equivalent interleaving of operations of transactions T and U

Transaction T: Transaction U:

x = read(i)

write(i, 10)

y = read(j)

write(j, 30)

write(j, 20)

z = read (i)704 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

operation, the timestamp of the transaction is compared with that of the object to

determine whether it can be done immediately or must be delayed or rejected. When an

operation is delayed, the transaction waits; when it is rejected, the transaction is aborted.

Basically, concurrency control can be achieved either by clients’ transactions

waiting for one another or by restarting transactions after conflicts between operations

have been detected, or by a combination of the two.

16.2.2 Recoverability from aborts

Servers must record all the effects of committed transactions and none of the effects of

aborted transactions. They must therefore allow for the fact that a transaction may abort

by preventing it affecting other concurrent transactions if it does so.

This section illustrates two problems associated with aborting transactions in the

context of the banking example. These problems are called ‘dirty reads’ and ‘premature

writes’, and both of them can occur in the presence of serially equivalent executions of

transactions. These issues are concerned with the effects of operations on objects such

as the balance of a bank account. To simplify things, operations are considered in two

categories: read operations and write operations. In our illustrations, getBalance is a

read operation and setBalance a write operation.

Dirty reads • The isolation property of transactions requires that transactions do not see

the uncommitted state of other transactions. The ‘dirty read’ problem is caused by the

interaction between a read operation in one transaction and an earlier write operation in

another transaction on the same object. Consider the executions illustrated in Figure

16.11, in which T gets the balance of account A and sets it to $10 more, then U gets the

balance of account A and sets it to $20 more, and the two executions are serially

equivalent. Now suppose that the transaction T aborts after U has committed. Then the

transaction U will have seen a value that never existed, since A will be restored to its

original value. We say that the transaction U has performed a dirty read. As it has

committed, it cannot be undone.

Figure 16.11 A dirty read when transaction T aborts

Transaction T: Transaction U:

a.getBalance()

a.setBalance(balance + 10)

a.getBalance()

a.setBalance(balance + 20)

balance = a.getBalance() $100

a.setBalance(balance + 10) $110

balance = a.getBalance() $110

a.setBalance(balance + 20) $130

commit transaction

abort transactionSECTION 16.2 TRANSACTIONS 705

Recoverability of transactions • If a transaction (like U) has committed after it has seen

the effects of a transaction that subsequently aborted, the situation is not recoverable. To

ensure that such situations will not arise, any transaction (like U) that is in danger of

having a dirty read delays its commit operation. The strategy for recoverability is to

delay commits until after the commitment of any other transaction whose uncommitted

state has been observed. In our example, U delays its commit until after T commits. In

the case that T aborts, then U must abort as well.

Cascading aborts • In Figure 16.11, suppose that transaction U delays committing until

after T aborts. As we have said, U must abort as well. Unfortunately, if any other

transactions have seen the effects due to U, they too must be aborted. The aborting of

these latter transactions may cause still further transactions to be aborted. Such

situations are called cascading aborts. To avoid cascading aborts, transactions are only

allowed to read objects that were written by committed transactions. To ensure that this

is the case, any read operation must be delayed until other transactions that applied a

write operation to the same object have committed or aborted. The avoidance of

cascading aborts is a stronger condition than recoverability.

Premature writes • Consider another implication of the possibility that a transaction

may abort. This one is related to the interaction between write operations on the same

object belonging to different transactions. For an illustration, we consider two

setBalance transactions, T and U, on account A, as shown in Figure 16.12. Before the

transactions, the balance of account A was $100. The two executions are serially

equivalent, with T setting the balance to $105 and U setting it to $110. If the transaction

U aborts and T commits, the balance should be $105.

Some database systems implement the action of abort by restoring ‘before

images’ of all the writes of a transaction. In our example, A is $100 initially, which is

the ‘before image’ of T’s write; similarly, $105 is the ‘before image’ of U’s write. Thus

if U aborts, we get the correct balance of $105.

Now consider the case when U commits and then T aborts. The balance should be

$110, but as the ‘before image’ of T’s write is $100, we get the wrong balance of $100.

Similarly, if T aborts and then U aborts, the ‘before image’ of U’s write is $105 and we

get the wrong balance of $105 – the balance should revert to $100.

To ensure correct results in a recovery scheme that uses before images, write

operations must be delayed until earlier transactions that updated the same objects have

either committed or aborted.

Figure 16.12 Overwriting uncommitted values

Transaction T: Transaction U:

a.setBalance(105) a.setBalance(110)

$100

a.setBalance(105) $105

a.setBalance(110) $110706 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

Strict executions of transactions • Generally, it is required that transactions delay both

their read and write operations so as to avoid both dirty reads and premature writes. The

executions of transactions are called strict if the service delays both read and write

operations on an object until all transactions that previously wrote that object have either

committed or aborted. The strict execution of transactions enforces the desired property

of isolation.

Tentative versions • For a server of recoverable objects to participate in transactions, it

must be designed so that any updates of objects can be removed if and when a

transaction aborts. To make this possible, all of the update operations performed during

a transaction are done in tentative versions of objects in volatile memory. Each

transaction is provided with its own private set of tentative versions of any objects that

it has altered. All the update operations of a transaction store values in the transaction’s

own private set. Access operations in a transaction take values from the transaction’s

own private set if possible, or failing that, from the objects.

The tentative versions are transferred to the objects only when a transaction

commits, by which time they will also have been recorded in permanent storage. This is

performed in a single step, during which other transactions are excluded from access to

the objects that are being altered. When a transaction aborts, its tentative versions are

deleted.

16.3 Nested transactions

Nested transactions extend the above transaction model by allowing transactions to be

composed of other transactions. Thus several transactions may be started from within a

transaction, allowing transactions to be regarded as modules that can be composed as

required.

The outermost transaction in a set of nested transactions is called the top-level

transaction. Transactions other than the top-level transaction are called subtransactions.

For example, in Figure 16.13, T is a top-level transaction that starts a pair of

subtransactions, T1 and T2. The subtransaction T1 starts its own pair of subtransactions,

T11 and T22. Also, subtransaction T2 starts its own subtransaction, T21, which starts

another subtransaction, T211.

A subtransaction appears atomic to its parent with respect to transaction failures

and to concurrent access. Subtransactions at the same level, such as T1 and T2, can run

concurrently, but their access to common objects is serialized – for example, by the

locking scheme described in Section 16.4. Each subtransaction can fail independently of

its parent and of the other subtransactions. When a subtransaction aborts, the parent

transaction can sometimes choose an alternative subtransaction to complete its task. For

example, a transaction to deliver a mail message to a list of recipients could be structured

as a set of subtransactions, each of which delivers the message to one of the recipients.

If one or more of the subtransactions fails, the parent transaction could record the fact

and then commit, with the result that all the successful child transactions commit. It

could then start another transaction to attempt to redeliver the messages that were not

sent the first time.SECTION 16.3 NESTED TRANSACTIONS 707

When we need to distinguish our original form of transaction from nested ones,

we use the term flat transaction. It is flat because all of its work is done at the same level

between an openTransaction and a commit or abort, and it is not possible to commit or

abort parts of it. Nested transactions have the following main advantages:

1. Subtransactions at one level (and their descendants) may run concurrently with

other subtransactions at the same level in the hierarchy. This can allow additional

concurrency in a transaction. When subtransactions run in different servers, they

can work in parallel. For example, consider the branchTotal operation in our

banking example. It can be implemented by invoking getBalance at every account

in the branch. Now each of these invocations may be performed as a

subtransaction, in which case they can be performed concurrently. Since each one

applies to a different account, there will be no conflicting operations among the

subtransactions.

2. Subtransactions can commit or abort independently. In comparison with a single

transaction, a set of nested subtransactions is potentially more robust. The above

example of delivering mail shows that this is so – with a flat transaction, one

transaction failure would cause the whole transaction to be restarted. In fact, a

parent can decide on different actions according to whether a subtransaction has

aborted or not.

The rules for committing of nested transactions are rather subtle:

• A transaction may commit or abort only after its child transactions have

completed.

• When a subtransaction completes, it makes an independent decision either to

commit provisionally or to abort. Its decision to abort is final.

• When a parent aborts, all of its subtransactions are aborted. For example, if T2

aborts then T21 and T211 must also abort, even though they may have provisionally

committed.

• When a subtransaction aborts, the parent can decide whether to abort or not. In our

example, T decides to commit although T2 has aborted.

Figure 16.13 Nested transactions

T : top-level transaction

T1 = openSubTransaction T2 = openSubTransaction

openSubTransaction openSubTransaction openSubTransaction

openSubTransaction

T1 : T2 :

T11 : T12 :

T211 :

T21 :

provisional commit

provisional commit

abort

provisional commit provisional commit

provisional commit

commit708 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

• If the top-level transaction commits, then all of the subtransactions that have

provisionally committed can commit too, provided that none of their ancestors has

aborted. In our example, T’s commitment allows T1, T11 and T12 to commit, but

not T21 and T211 since their parent, T2, aborted. Note that the effects of a

subtransaction are not permanent until the top-level transaction commits.

In some cases, the top-level transaction may decide to abort because one or more of its

subtransactions have aborted. As an example, consider the following Transfer

transaction:

Transfer $100 from B to A

a.deposit(100)

b.withdraw(100)

This can be structured as a pair of subtransactions, one for the withdraw operation and

the other for deposit. When the two subtransactions both commit, the Transfer

transaction can also commit. Suppose that a withdraw subtransaction aborts whenever

an account is overdrawn. Now consider the case when the withdraw subtransaction

aborts and the deposit subtransaction commits – and recall that the commitment of a

child transaction is conditional on the parent transaction committing. We presume that

the top-level (Transfer) transaction will decide to abort. The aborting of the parent

transaction causes the subtransactions to abort – so the deposit transaction is aborted and

all its effects are undone.

The CORBA Object Transaction Service supports both flat and nested

transactions. Nested transactions are particularly useful in distributed systems because

child transactions may be run concurrently in different servers. We return to this issue

in Chapter 17. This form of nested transactions is due to Moss [1985]. Other variants of

nested transactions with different serializability properties have been proposed; for

example, see Weikum [1991].

16.4 Locks

Transactions must be scheduled so that their effect on shared data is serially equivalent.

A server can achieve serial equivalence of transactions by serializing access to the

objects. Figure 16.7 shows an example of how serial equivalence can be achieved with

some degree of concurrency – transactions T and U both access account B, but T

completes its access before U starts accessing it.

A simple example of a serializing mechanism is the use of exclusive locks. In this

locking scheme, the server attempts to lock any object that is about to be used by any

operation of a client’s transaction. If a client requests access to an object that is already

locked due to another client’s transaction, the request is suspended and the client must

wait until the object is unlocked.

Figure 16.14 illustrates the use of exclusive locks. It shows the same transactions

as Figure 16.7, but with an extra column for each transaction showing the locking,

waiting and unlocking. In this example, it is assumed that when transactions T and U

start, the balances of the accounts A, B and C are not yet locked. When transaction T is

about to use account B, it is locked for T. When transaction U is about to use B it is stillSECTION 16.4 LOCKS 709

locked for T, so transaction U waits. When transaction T is committed, B is unlocked,

whereupon transaction U is resumed. The use of the lock on B effectively serializes the

access to B. Note that if, for example, T released the lock on B between its getBalance

and setBalance operations, transaction U’s getBalance operation on B could be

interleaved between them.

Serial equivalence requires that all of a transaction’s accesses to a particular object

be serialized with respect to accesses by other transactions. All pairs of conflicting

operations of two transactions should be executed in the same order. To ensure this, a

transaction is not allowed any new locks after it has released a lock. The first phase of

each transaction is a ‘growing phase’, during which new locks are acquired. In the

second phase, the locks are released (a ‘shrinking phase’). This is called two-phase

locking.

We saw in Section 16.2.2 that because transactions may abort, strict executions

are needed to prevent dirty reads and premature writes. Under a strict execution regime,

a transaction that needs to read or write an object must be delayed until other

transactions that wrote the same object have committed or aborted. To enforce this rule,

any locks applied during the progress of a transaction are held until the transaction

commits or aborts. This is called strict two-phase locking. The presence of the locks

prevents other transactions reading or writing the objects. When a transaction commits,

to ensure recoverability, the locks must be held until all the objects it updated have been

written to permanent storage.

A server generally contains a large number of objects, and a typical transaction

accesses only a few of them and is unlikely to clash with other current transactions. The

granularity with which concurrency control can be applied to objects is an important

Figure 16.14 Transactions T and U with exclusive locks

Transaction T: Transaction U:

balance = b.getBalance()

b.setBalance(bal\*1.1)

a.withdraw(bal/10)

balance = b.getBalance()

b.setBalance(bal\*1.1)

c.withdraw(bal/10)

Operations Locks Operations Locks

openTransaction

bal = b.getBalance() lock B

b.setBalance(bal\*1.1) openTransaction

a.withdraw(bal/10) lock A bal = b.getBalance() waits for T’s

lock on B

closeTransaction unlock A, B • • •

lock B

b.setBalance(bal\*1.1)

c.withdraw(bal/10) lock C

closeTransaction unlock B, C710 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

issue, since the scope for concurrent access to objects in a server will be limited severely

if concurrency control (for example, locks) can only be applied to all the objects at once.

In our banking example, if locks were applied to all customer accounts at a branch, only

one bank clerk could perform an online banking transaction at any time – hardly an

acceptable constraint!

The portion of the objects to which access must be serialized should be as small

as possible; that is, just that part involved in each operation requested by transactions.

In our banking example, a branch holds a set of accounts, each of which has a balance.

Each banking operation affects one or more account balances – deposit and withdraw

affect one account balance, and branchTotal affects all of them.

The description of concurrency control schemes given below does not assume any

particular granularity. We discuss concurrency control protocols that are applicable to

objects whose operations can be modelled in terms of read and write operations on the

objects. For the protocols to work correctly, it is essential that each read and write

operation is atomic in its effects on objects.

Concurrency control protocols are designed to cope with conflicts between

operations in different transactions on the same object. In this chapter, we use the notion

of conflict between operations to explain the protocols. The conflict rules for read and

write operations are given in Figure 16.9, which shows that pairs of read operations

from different transactions on the same object do not conflict. Therefore, a simple

exclusive lock that is used for both read and write operations reduces concurrency more

than is necessary.

It is preferable to adopt a locking scheme that controls the access to each object so

that there can be several concurrent transactions reading an object, or a single

transaction writing an object, but not both. This is commonly referred to as a ‘many

readers/single writer’ scheme. Two types of locks are used: read locks and write locks.

Before a transaction’s read operation is performed, a read lock should be set on the

object. Before a transaction’s write operation is performed, a write lock should be set on

the object. Whenever it is impossible to set a lock immediately, the transaction (and the

client) must wait until it is possible to do so – a client’s request is never rejected.

As pairs of read operations from different transactions do not conflict, an attempt

to set a read lock on an object with a read lock is always successful. All the transactions

reading the same object share its read lock – for this reason, read locks are sometimes

called shared locks.

The operation conflict rules tell us that:

1. If a transaction T has already performed a read operation on a particular object,

then a concurrent transaction U must not write that object until T commits or

aborts.

2. If a transaction T has already performed a write operation on a particular object,

then a concurrent transaction U must not read or write that object until T commits

or aborts.

To enforce condition 1, a request for a write lock on an object is delayed by the presence

of a read lock belonging to another transaction. To enforce condition 2, a request for

either a read lock or a write lock on an object is delayed by the presence of a write lock

belonging to another transaction.SECTION 16.4 LOCKS 711

Figure 16.15 shows the compatibility of read locks and write locks on any

particular object. The entries to the left of the first column in the table show the type of

lock already set, if any. The entries above the first row show the type of lock requested.

The entry in each cell shows the effect on a transaction that requests the type of lock

given above when the object has been locked in another transaction with the type of lock

on the left.

Inconsistent retrievals and lost updates are caused by conflicts between read

operations in one transaction and write operations in another without the protection of a

concurrency control scheme such as locking. Inconsistent retrievals are prevented by

performing the retrieval transaction before or after the update transaction. If the retrieval

transaction comes first, its read locks delay the update transaction. If it comes second,

its request for read locks causes it to be delayed until the update transaction has

completed.

Lost updates occur when two transactions read a value of an object and then use

it to calculate a new value. Lost updates are prevented by making later transactions delay

their reads until the earlier ones have completed. This is achieved by each transaction

setting a read lock when it reads an object and then promoting it to a write lock when it

writes the same object – when a subsequent transaction requires a read lock it will be

delayed until any current transaction has completed.

A transaction with a read lock that is shared with other transactions cannot

promote its read lock to a write lock, because the latter would conflict with the read locks

held by the other transactions. Therefore, such a transaction must request a write lock

and wait for the other read locks to be released.

Lock promotion refers to the conversion of a lock to a stronger lock – that is, a

lock that is more exclusive. The lock compatibility table in Figure 16.15 shows the

relative exclusivity of locks. The read lock allows other read locks, whereas the write

lock does not. Neither allows other write locks. Therefore, a write lock is more exclusive

than a read lock. Locks may be promoted because the result is a more exclusive lock. It

is not safe to demote a lock held by a transaction before it commits, because the result

will be more permissive than the previous one and may allow executions by other

transactions that are inconsistent with serial equivalence.

The rules for the use of locks in a strict two-phase locking implementation are

summarized in Figure 16.16. To ensure that these rules are adhered to, the client has no

access to operations for locking or unlocking items of data. Locking is performed when

the requests for read and write operations are about to be applied to the recoverable

objects, and unlocking is performed by the commit or abort operations of the transaction

coordinator.

Figure 16.15 Lock compatibility

For one object Lock requested

read write

Lock already set none OK OK

read OK wait

write wait wait712 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

For example, the CORBA Concurrency Control Service [OMG 2000b] can be

used to apply concurrency control on behalf of transactions or to protect objects without

using transactions. It provides a means of associating a collection of locks (called a

lockset) with a resource such as a recoverable object. A lockset allows locks to be

acquired or released. A lockset’s lock method will acquire a lock or block until the lock

is free; other methods allow locks to be promoted or released. Transactional locksets

support the same methods as locksets, but their methods require transaction identifiers

as arguments. We mentioned earlier that the CORBA transaction service tags all client

requests in a transaction with the transaction identifier. This enables a suitable lock to

be acquired before each of the recoverable objects is accessed during a transaction. The

transaction coordinator is responsible for releasing the locks when a transaction

commits or aborts.

The rules given in Figure 16.16 ensure strictness, because the locks are held until

a transaction has either committed or aborted. However, it is not necessary to hold read

locks to ensure strictness. Read locks need only be held until the request to commit or

abort arrives.

Lock implementation • The granting of locks will be implemented by a separate object

in the server that we call the lock manager. The lock manager holds a set of locks, for

example in a hash table. Each lock is an instance of the class Lock and is associated with

a particular object. The class Lock is shown in Figure 16.17. Each instance of Lock

maintains the following information in its instance variables:

• the identifier of the locked object;

• the transaction identifiers of the transactions that currently hold the lock (shared

locks can have several holders);

• a lock type.

Figure 16.16 Use of locks in strict two-phase locking

1. When an operation accesses an object within a transaction:

(a)If the object is not already locked, it is locked and the operation proceeds.

(b)If the object has a conflicting lock set by another transaction, the transaction

must wait until it is unlocked.

(c)If the object has a non-conflicting lock set by another transaction, the lock is

shared and the operation proceeds.

(d)If the object has already been locked in the same transaction, the lock will be

promoted if necessary and the operation proceeds. (Where promotion is

prevented by a conflicting lock, rule b is used.)

2. When a transaction is committed or aborted, the server unlocks all objects it

locked for the transaction.SECTION 16.4 LOCKS 713

The methods of Lock are synchronized so that the threads attempting to acquire or

release a lock will not interfere with one another. But, in addition, attempts to acquire

the lock use the wait method whenever they have to wait for another thread to release it.

The acquire method carries out the rules given in Figure 16.15 and Figure 16.16.

Its arguments specify a transaction identifier and the type of lock required by that

transaction. It tests whether the request can be granted. If another transaction holds the

lock in a conflicting mode, it invokes wait, which causes the caller’s thread to be

suspended until a corresponding notify. Note that the wait is enclosed in a while, because

all waiters are notified and some of them may not be able to proceed. When, eventually,

the condition is satisfied, the remainder of the method sets the lock appropriately:

• if no other transaction holds the lock, just add the given transaction to the holders

and set the type;

• else if another transaction holds the lock, share it by adding the given transaction

to the holders (unless it is already a holder);

• else if this transaction is a holder but is requesting a more exclusive lock, promote

the lock.

Figure 16.17 Lock class

public class Lock {

private Object object; // the object being protected by the lock

private Vector holders; // the TIDs of current holders

private LockType lockType; // the current type

public synchronized void acquire(TransID trans, LockType aLockType ){

while(/\*another transaction holds the lock in conflicting mode\*/) {

try {

wait();

}catch ( InterruptedException e){/\*...\*/ }

}

if (holders.isEmpty()) { // no TIDs hold lock

holders.addElement(trans);

lockType = aLockType;

} else if (/\*another transaction holds the lock, share it\*/ ) ){

if (/\* this transaction not a holder\*/) holders.addElement(trans);

} else if (/\* this transaction is a holder but needs a more exclusive lock\*/)

lockType.promote();

}

}

public synchronized void release(TransID trans ){

holders.removeElement(trans); // remove this holder

// set locktype to none

notifyAll();

}

}714 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

The release method’s arguments specify the transaction identifier of the transaction that

is releasing the lock. It removes the transaction identifier from the holders, sets the lock

type to none and calls notifyAll. The method notifies all waiting threads in case there are

multiple transactions waiting to acquire read locks – all of them may be able to proceed.

The class LockManager is shown in Figure 16.18. All requests to set locks and to release

them on behalf of transactions are sent to an instance of LockManager:

• The setLock method’s arguments specify the object that the given transaction

wants to lock and the type of lock. It finds a lock for that object in its hashtable or,

if necessary, creates one. It then invokes the acquire method of that lock.

• The unLock method’s argument specifies the transaction that is releasing its locks.

It finds all of the locks in the hashtable that have the given transaction as a holder.

For each one, it calls the release method.

Some questions of policy: Note that when several threads wait on the same locked item,

the semantics of wait ensure that each transaction gets its turn. In the above program, the

conflict rules allow the holders of a lock to be either multiple readers or one writer. The

arrival of a request for a read lock is always granted unless the holder has a write lock.

The reader is invited to consider the following:

What is the consequence for write transactions in the presence of a steady trickle

of requests for read locks? Think of an alternative implementation.

Figure 16.18 LockManager class

public class LockManager {

private Hashtable theLocks;

public void setLock(Object object, TransID trans, LockType lockType){

Lock foundLock;

synchronized(this){

// find the lock associated with object

// if there isn’t one, create it and add it to the hashtable

}

foundLock.acquire(trans, lockType);

}

// synchronize this one because we want to remove all entries

public synchronized void unLock(TransID trans) {

Enumeration e = theLocks.elements();

while(e.hasMoreElements()){

Lock aLock = (Lock)(e.nextElement());

if(/\* trans is a holder of this lock\*/ ) aLock.release(trans);

}

}

}SECTION 16.4 LOCKS 715

When the holder has a write lock, several readers and writers may be waiting. The

reader should consider the effect of notifyAll and think of an alternative

implementation. If a holder of a read lock tries to promote the lock when the lock

is shared, it will be blocked. Is there any solution to this difficulty?

Locking rules for nested transactions • The aim of a locking scheme for nested

transactions is to serialize access to objects so that:

1. Each set of nested transactions is a single entity that must be prevented from

observing the partial effects of any other set of nested transactions.

2. Each transaction within a set of nested transactions must be prevented from

observing the partial effects of the other transactions in the set.

The first rule is enforced by arranging that every lock that is acquired by a successful

subtransaction is inherited by its parent when it completes. Inherited locks are also

inherited by ancestors. Note that this form of inheritance passes from child to parent!

The top-level transaction eventually inherits all of the locks that were acquired by

successful subtransactions at any depth in a nested transaction. This ensures that the

locks can be held until the top-level transaction has committed or aborted, which

prevents members of different sets of nested transactions observing one another’s partial

effects.

The second rule is enforced as follows:

• Parent transactions are not allowed to run concurrently with their child

transactions. If a parent transaction has a lock on an object, it retains the lock

during the time that its child transaction is executing. This means that the child

transaction temporarily acquires the lock from its parent for its duration.

• Subtransactions at the same level are allowed to run concurrently, so when they

access the same objects, the locking scheme must serialize their access.

The following rules describe lock acquisition and release:

• For a subtransaction to acquire a read lock on an object, no other active transaction

can have a write lock on that object, and the only retainers of a write lock are its

ancestors.

• For a subtransaction to acquire a write lock on an object, no other active

transaction can have a read or write lock on that object, and the only retainers of

read and write locks on that object are its ancestors.

• When a subtransaction commits, its locks are inherited by its parent, allowing the

parent to retain the locks in the same mode as the child.

• When a subtransaction aborts, its locks are discarded. If the parent already retains

the locks, it can continue to do so.

Note that subtransactions at the same level that access the same object will take turns to

acquire the locks retained by their parent. This ensures that their access to a common

object is serialized.

As an example, suppose that subtransactions T1, T2 and T11 in Figure 16.13 all

access a common object, which is not accessed by the top-level transaction T. Suppose

that subtransaction T1 is the first to access the object and successfully acquires a lock,716 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

which it passes on to T11 for the duration of its execution, getting it back when T11

completes. When T1 completes its execution, the top-level transaction T inherits the

lock, which it retains until the set of nested transactions completes. The subtransaction

T2 can acquire the lock from T for the duration of its execution.

16.4.1 Deadlocks

The use of locks can lead to deadlock. Consider the use of locks shown in Figure 16.19.

Since the deposit and withdraw methods are atomic, we show them acquiring write locks

– although in practice they read the balance and then write it. Each of them acquires a

lock on one account and then gets blocked when it tries to access the account that the

other one has locked. This is a deadlock situation – two transactions are waiting, and

each is dependent on the other to release a lock so it can resume.

Deadlock is a particularly common situation when clients are involved in an

interactive program, for a transaction in an interactive program may last for a long

period of time. This can result in many objects being locked and remaining so, thus

preventing other clients using them.

Note that the locking of subitems in structured objects can be useful for avoiding

conflicts and possible deadlock situations. For example, a day in a diary could be

structured as a set of timeslots, each of which can be locked independently for updating.

Hierarchic locking schemes are useful if the application requires a different granularity

of locking for different operations, see Section 16.4.2.

Definition of deadlock • Deadlock is a state in which each member of a group of

transactions is waiting for some other member to release a lock. A wait-for graph can

be used to represent the waiting relationships between current transactions. In a wait-for

graph the nodes represent transactions and the edges represent wait-for relationships

between transactions – there is an edge from node T to node U when transaction T is

waiting for transaction U to release a lock. Figure 16.20 illustrates the wait-for graph

corresponding to the deadlock situation illustrated in Figure 16.19. Recall that the

deadlock arose because transactions T and U both attempted to acquire an object held by

the other. Therefore T waits for U and U waits for T. The dependency between

Figure 16.19 Deadlock with write locks

Transaction T Transaction U

Operations Locks Operations Locks

a.deposit(100); write lock A

b.deposit(200) write lock B

b.withdraw(100)

••• waits for U’s a.withdraw(200); waits for T’s

lock on B ••• lock on A

••• •••

••• •••SECTION 16.4 LOCKS 717

transactions is indirect, via a dependency on objects. The diagram on the right shows the

objects held by and waited for by transactions T and U. As each transaction can wait for

only one object, the objects can be omitted from the wait-for graph – leaving the simple

graph on the left.

Suppose that, as in Figure 16.21, a wait-for graph contains a cycle T → U → …

→ V → T . Each transaction is waiting for the next transaction in the cycle. All of these

transactions are blocked waiting for locks. None of the locks can ever be released, and

the transactions are deadlocked. If one of the transactions in a cycle is aborted, then its

locks are released and that cycle is broken. For example, if transaction T in Figure 16.21

is aborted, it will release a lock on an object that V is waiting for – and V will no longer

be waiting for T.

Now consider a scenario in which the three transactions T, U and V share a read

lock on an object C, and transaction W holds a write lock on object B, on which

transaction V is waiting to obtain a lock (as shown on the right in Figure 16.22). The

transactions T and W then request write locks on object C, and a deadlock situation arises

in which T waits for U and V, V waits for W, and W waits for T, U and V, as shown on

the left in Figure 16.22. This shows that although each transaction can wait for only one

object at a time, it may be involved in several cycles. For example, transaction V is

involved in two cycles: V → W → T → V and V → W → V.

In this example, suppose that transaction V is aborted. This will release V’s lock

on C and the two cycles involving V will be broken.

Deadlock prevention • One solution is to prevent deadlock. An apparently simple but

not very good way to overcome the deadlock problem is to lock all of the objects used

by a transaction when it starts. This would need to be done as a single atomic step so as

Figure 16.20 The wait-for graph for Figure 16.19

A B

Waits for

Held by

Held by

T U T U

Waits for

Figure 16.21 A cycle in a wait-for graph

U

V

T718 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

to avoid deadlock at this stage. Such a transaction cannot run into deadlocks with other

transactions, but this approach unnecessarily restricts access to shared resources. In

addition, it is sometimes impossible to predict at the start of a transaction which objects

will be used. This is generally the case in interactive applications, for the user would

have to say in advance exactly which objects they were planning to use – this is

inconceivable in browsing-style applications, which allow users to find objects they do

not know about in advance. Deadlocks can also be prevented by requesting locks on

objects in a predefined order, but this can result in premature locking and a reduction in

concurrency.

Upgrade locks • CORBA’s Concurrency Control Service introduces a third type of

lock, called upgrade, the use of which is intended to avoid deadlocks. Deadlocks are

often caused by two conflicting transactions first taking read locks and then attempting

to promote them to write locks. A transaction with an upgrade lock on a data item is

permitted to read that data item, but this lock conflicts with any upgrade locks set by

other transactions on the same data item. This type of lock cannot be set implicitly by

the use of a read operation, but must be requested by the client.

Deadlock detection • Deadlocks may be detected by finding cycles in the wait-for

graph. Having detected a deadlock, a transaction must be selected for abortion to break

the cycle.

The software responsible for deadlock detection can be part of the lock manager.

It must hold a representation of the wait-for graph so that it can check it for cycles from

time to time. Edges are added to the graph and removed from the graph by the lock

manager’s setLock and unLock operations. At the point illustrated by Figure 16.22 on

the left, it will have the following information:

An edge T → U is added whenever the lock manager blocks a request by transaction T

for a lock on an object that is already locked on behalf of transaction U. Note that when

Transaction Waits for transaction

T U, V

V W

W T, U, V

C

T

U

V

Held by

Held by

Held by

T

U

V

W

W

B

Held by

Waits for

Figure 16.22 Another wait-for graphSECTION 16.4 LOCKS 719

a lock is shared, several edges may be added. An edge T → U is deleted whenever U

releases a lock that T is waiting for and allows T to proceed. See Exercise 16.14 for a

more detailed discussion of the implementation of deadlock detection. If a transaction

shares a lock, the lock is not released, but the edges leading to a particular transaction

are removed.

The presence of cycles may be checked each time an edge is added, or less

frequently to avoid unnecessary overhead. When a deadlock is detected, one of the

transactions in the cycle must be chosen and then be aborted. The corresponding node

and the edges involving it must be removed from the wait-for graph. This will happen

when the aborted transaction has its locks removed.

The choice of the transaction to abort is not simple. Some factors that may be

taken into account are the age of the transaction and the number of cycles in which it is

involved.

Timeouts • Lock timeouts are a method for resolution of deadlocks that is commonly

used. Each lock is given a limited period in which it is invulnerable. After this time, a

lock becomes vulnerable. Provided that no other transaction is competing for the object

that is locked, an object with a vulnerable lock remains locked. However, if any other

transaction is waiting to access the object protected by a vulnerable lock, the lock is

broken (that is, the object is unlocked) and the waiting transaction resumes. The

transaction whose lock has been broken is normally aborted.

There are many problems with the use of timeouts as a remedy for deadlocks: the

worst problem is that transactions are sometimes aborted due to their locks becoming

vulnerable when other transactions are waiting for them, but there is actually no

deadlock. In an overloaded system, the number of transactions timing out will increase,

and transactions taking a long time can be penalized. In addition, it is hard to decide on

an appropriate length for a timeout. In contrast, if deadlock detection is used,

Figure 16.23 Resolution of the deadlock in Figure 16.19

Transaction T Transaction U

Operations Locks Operations Locks

a.deposit(100); write lock A

b.deposit(200) write lock B

b.withdraw(100)

••• waits for U’s a.withdraw(200); waits for T’s

lock on B ••• lock on A

(timeout elapses)

T’s lock on A becomes vulnerable,

unlock A, abort T

•••

a.withdraw(200); write lock A

unlock A, B720 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

transactions are aborted because deadlocks have occurred and a choice can be made as

to which transaction to abort.

Using lock timeouts, we can resolve the deadlock in Figure 16.19 as shown in

Figure 16.23, in which the write lock for T on A becomes vulnerable after its timeout

period. Transaction U is waiting to acquire a write lock on A. Therefore, T is aborted and

it releases its lock on A, allowing U to resume and complete the transaction.

When transactions access objects located in several different servers, the

possibility of distributed deadlocks arises. In a distributed deadlock, the wait-for graph

can involve objects at multiple locations. We return to this subject in Section 17.5.

16.4.2 Increasing concurrency in locking schemes

Even when locking rules are based on the conflicts between read and write operations

and the granularity at which they are applied is as small as possible, there is still some

scope for increasing concurrency. We discuss two approaches that have been used to

deal with this issue. In the first approach (two-version locking), the setting of exclusive

locks is delayed until a transaction commits. In the second approach (hierarchic locks),

mixed-granularity locks are used.

Two-version locking • This is an optimistic scheme that allows one transaction to write

tentative versions of objects while other transactions read from the committed versions

of the same objects. read operations only wait if another transaction is currently

committing the same object. This scheme allows more concurrency than read-write

locks, but writing transactions risk waiting or even rejection when they attempt to

commit. Transactions cannot commit their write operations immediately if other

uncompleted transactions have read the same objects. Therefore, transactions that

request to commit in such a situation are made to wait until the reading transactions have

completed. Deadlocks may occur when transactions are waiting to commit. Therefore,

transactions may need to be aborted when they are waiting to commit, to resolve

deadlocks.

This variation on strict two-phase locking uses three types of lock: a read lock, a

write lock and a commit lock. Before a transaction’s read operation is performed, a read

lock must be set on the object – the attempt to set a read lock is successful unless the

object has a commit lock, in which case the transaction waits. Before a transaction’s

Figure 16.24 Lock compatibility (read, write and commit locks)

For one object Lock to be set

read write commit

Lock already set none OK OK OK

read OK OK wait

write OK wait –

commit wait wait –SECTION 16.4 LOCKS 721

write operation is performed, a write lock must be set on the object – the attempt to set

a write lock is successful unless the object has a write lock or a commit lock, in which

case the transaction waits.

When the transaction coordinator receives a request to commit a transaction, it

attempts to convert all that transaction’s write locks to commit locks. If any of the

objects have outstanding read locks, the transaction must wait until the transactions that

set these locks have completed and the locks are released. The compatibility of read,

write and commit locks is shown in Figure 16.24.

There are two main differences in performance between the two-version locking

scheme and an ordinary read-write locking scheme. On the one hand, read operations in

the two-version locking scheme are delayed only while the transactions are being

committed, rather than during the entire execution of transactions – in most cases, the

commit protocol takes only a small fraction of the time required to perform an entire

transaction. On the other hand, read operations of one transaction can cause delays in

committing other transactions.

Hierarchic locks • In some applications, the granularity suitable for one operation is not

appropriate for another operation. In our banking example, the majority of the

operations require locking at the granularity of an account. The branchTotal operation

is different – it reads the values of all the account balances and would appear to require

a read lock on all of them. To reduce locking overhead, it would be useful to allow locks

of mixed granularity to coexist.

Gray [1978] proposed the use of a hierarchy of locks with different granularities.

At each level, the setting of a parent lock has the same effect as setting all the equivalent

child locks. This economizes on the number of locks to be set. In our banking example,

the branch is the parent and the accounts are children (see Figure 16.25).

Mixed-granularity locks could be useful in a diary system in which the data could

be structured with the diary for a week being composed of a page for each day and the

Branch

A B C Account

Figure 16.25 Lock hierarchy for the banking example

Figure 16.26 Lock hierarchy for a diary

Week

Monday Tuesday Wednesday Thursday Friday

9:00–10:00

timeslots

10:00–11:00 11:00–12:00 12:00–13:00 13:00–14:00 14:00–15:00 15:00–16:00722 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

latter subdivided further into a slot for each hour of the day, as shown in Figure 16.26.

The operation to view a week would cause a read lock to be set at the top of this

hierarchy, whereas the operation to enter an appointment would cause a write lock to be

set on a given time slot. The effect of a read lock on a week would be to prevent write

operations on any of the substructures – for example, the time slots for each day in that

week.

In Gray’s scheme, each node in the hierarchy can be locked, giving the owner of

the lock explicit access to the node and giving implicit access to its children. In our

example, in Figure 16.25 a read-write lock on the branch implicitly read-write locks all

the accounts. Before a child node is granted a read-write lock, an intention to read-write

lock is set on the parent node and its ancestors (if any). The intention lock is compatible

with other intention locks but conflicts with read and write locks according to the usual

rules. Figure 16.27 gives the compatibility table for hierarchic locks. Gray also proposed

a third type of intention lock – one that combines the properties of a read lock with an

intention to write lock.

In our banking example, the branchTotal operation requests a read lock on the

branch, which implicitly sets read locks on all the accounts. A deposit operation needs

to set a write lock on a balance, but first it attempts to set an intention to write lock on

the branch. These rules prevent these operations running concurrently.

Hierarchic locks have the advantage of reducing the number of locks when mixedgranularity locking is required. The compatibility tables and the rules for promoting

locks are more complex.

The mixed granularity of locks could allow each transaction to lock a portion

whose size is chosen according to its needs. A long transaction that accesses many

objects could lock the whole collection, whereas a short transaction can lock at finer

granularity.

The CORBA Concurrency Control Service supports variable-granularity locking

with intention to read and intention to write lock types. These can be used as described

above to take advantage the opportunity to apply locks at differing granularities in

hierarchically structured data.

Figure 16.27 Lock compatibility table for hierarchic locks

For one object Lock to be set

read write I-read I-write

Lock already set none OK OK OK OK

read OK wait OK wait

write wait wait wait wait

I-read OK wait OK OK

I-write wait wait OK OKSECTION 16.5 OPTIMISTIC CONCURRENCY CONTROL 723

16.5 Optimistic concurrency control

Kung and Robinson [1981] identified a number of inherent disadvantages of locking and

proposed an alternative optimistic approach to the serialization of transactions that

avoids these drawbacks. We can summarize the drawbacks of locking:

• Lock maintenance represents an overhead that is not present in systems that do not

support concurrent access to shared data. Even read-only transactions (queries),

which cannot possibly affect the integrity of the data, must, in general, use locking

in order to guarantee that the data being read is not modified by other transactions

at the same time. But locking may be necessary only in the worst case.

For example, consider two client processes that are concurrently incrementing the

values of n objects. If the client programs start at the same time and run for about

the same amount of time, accessing the objects in two unrelated sequences and

using a separate transaction to access and increment each item, the chances that

the two programs will attempt to access the same object at the same time are just

1 in n on average, so locking is really needed only once in every n transactions.

• The use of locks can result in deadlock. Deadlock prevention reduces concurrency

severely, and therefore deadlock situations must be resolved either by the use of

timeouts or by deadlock detection. Neither of these is wholly satisfactory for use

in interactive programs.

• To avoid cascading aborts, locks cannot be released until the end of the

transaction. This may reduce significantly the potential for concurrency.

The alternative approach proposed by Kung and Robinson is ‘optimistic’ because it is

based on the observation that, in most applications, the likelihood of two clients’

transactions accessing the same object is low. Transactions are allowed to proceed as

though there were no possibility of conflict with other transactions until the client

completes its task and issues a closeTransaction request. When a conflict arises, some

transaction is generally aborted and will need to be restarted by the client. Each

transaction has the following phases:

Working phase: During the working phase, each transaction has a tentative version

of each of the objects that it updates. This is a copy of the most recently committed

version of the object. The use of tentative versions allows the transaction to abort

(with no effect on the objects), either during the working phase or if it fails validation

due to other conflicting transactions. read operations are performed immediately – if

a tentative version for that transaction already exists, a read operation accesses it;

otherwise, it accesses the most recently committed value of the object. write

operations record the new values of the objects as tentative values (which are

invisible to other transactions). When there are several concurrent transactions,

several different tentative values of the same object may coexist. In addition, two

records are kept of the objects accessed within a transaction: a read set containing the

objects read by the transaction and a write set containing the objects written by the

transaction. Note that as all read operations are performed on committed versions of

the objects (or copies of them), dirty reads cannot occur.724 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

Validation phase: When the closeTransaction request is received, the transaction is

validated to establish whether or not its operations on objects conflict with operations

of other transactions on the same objects. If the validation is successful, then the

transaction can commit. If the validation fails, then some form of conflict resolution

must be used and either the current transaction or, in some cases, those with which it

conflicts will need to be aborted.

Update phase: If a transaction is validated, all of the changes recorded in its tentative

versions are made permanent. Read-only transactions can commit immediately after

passing validation. Write transactions are ready to commit once the tentative versions

of the objects have been recorded in permanent storage.

Validation of transactions • Validation uses the read-write conflict rules to ensure that

the scheduling of a particular transaction is serially equivalent with respect to all other

overlapping transactions – that is, any transactions that had not yet committed at the time

this transaction started. To assist in performing validation, each transaction is assigned

a transaction number when it enters the validation phase (that is, when the client issues

a closeTransaction). If the transaction is validated and completes successfully, it retains

this number; if it fails the validation checks and is aborted, or if the transaction is read

only, the number is released for reassignment. Transaction numbers are integers

assigned in ascending sequence; the number of a transaction therefore defines its

position in time – a transaction always finishes its working phase after all transactions

with lower numbers. That is, a transaction with the number Ti always precedes a

transaction with the number Tj if i < j. (If the transaction number were to be assigned at

the beginning of the working phase, then a transaction that reached the end of the

working phase before one with a lower number would have to wait until the earlier one

had completed before it could be validated.)

The validation test on transaction Tv is based on conflicts between operations in

pairs of transactions Ti and Tv. For a transaction Tv to be serializable with respect to an

overlapping transaction Ti, their operations must conform to the following rules:

As the validation and update phases of a transaction are generally short in duration

compared with the working phase, a simplification can be achieved by making the rule

that only one transaction may be in the validation and update phase at one time. When

no two transactions may overlap in the update phase, rule 3 is satisfied. Note that this

restriction on write operations, together with the fact that no dirty reads can occur,

produces strict executions. To prevent overlapping, the entire validation and update

phases can be implemented as a critical section so that only one client at a time can

execute it. In order to increase concurrency, part of the validation and updating may be

Tv Ti Rule

write read 1. Ti must not read objects written by Tv.

read write 2. Tv must not read objects written by Ti.

write write 3. Ti must not write objects written by Tv and

Tv must not write objects written by Ti.SECTION 16.5 OPTIMISTIC CONCURRENCY CONTROL 725

implemented outside the critical section, but it is essential that the assignment of

transaction numbers is performed sequentially. We note that at any instant, the current

transaction number is like a pseudo-clock that ticks whenever a transaction completes

successfully.

The validation of a transaction must ensure that rules 1 and 2 are obeyed by testing

for overlaps between the objects of pairs of transactions Tv and Ti. There are two forms

of validation – backward and forward [Härder 1984]. Backward validation checks the

transaction undergoing validation with other preceding overlapping transactions – those

that entered the validation phase before it. Forward validation checks the transaction

undergoing validation with other later transactions, which are still active.

Backward validation • As all the read operations of earlier overlapping transactions

were performed before the validation of Tv started, they cannot be affected by the writes

of the current transaction (and rule 1 is satisfied). The validation of transaction Tv checks

whether its read set (the objects affected by the read operations of Tv) overlaps with any

of the write sets of earlier overlapping transactions, Ti (rule 2). If there is any overlap,

the validation fails.

Let startTn be the biggest transaction number assigned (to some other committed

transaction) at the time when transaction Tv started its working phase and finishTn be the

biggest transaction number assigned at the time when Tv entered the validation phase.

The following program describes the algorithm for the validation of Tv:

boolean valid = true;

for (int Ti = startTn+1; Ti <= finishTn; Ti++){

if (read set of Tv intersects write set of Ti) valid = false;

}

Figure 16.28 shows overlapping transactions that might be considered in the validation

of a transaction Tv. Time increases from left to right. The earlier committed transactions

are T1, T2 and T3. T1 committed before Tv started. T2 and T3 committed before Tv

finished its working phase. StartTn + 1 = T2 and finishTn = T3. In backward validation,

the read set of Tv must be compared with the write sets of T2 and T3.

Figure 16.28 Validation of transactions

Earlier committed

transactions

Working Validation Update

T1

Tv

Transaction

being validated

T2

T3

Later active

transactions

active1

active2726 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

In backward validation, the read set of the transaction being validated is compared

with the write sets of other transactions that have already committed. Therefore, the only

way to resolve any conflicts is to abort the transaction that is undergoing validation.

In backward validation, transactions that have no read operations (only write

operations) need not be checked.

Optimistic concurrency control with backward validation requires that the write

sets of old committed versions of objects corresponding to recently committed

transactions are retained until there are no unvalidated overlapping transactions with

which they might conflict. Whenever a transaction is successfully validated, its

transaction number, startTn and write set are recorded in a preceding transactions list that

is maintained by the transaction service. Note that this list is ordered by transaction

number. In an environment with long transactions, the retention of old write sets of

objects may be a problem. For example, in Figure 16.28 the write sets of T1, T2, T3 and

Tv must be retained until the active transaction active1 completes. Note that the although

the active transactions have transaction identifiers, they do not yet have transaction

numbers.

Forward validation • In forward validation of the transaction Tv, the write set of Tv is

compared with the read sets of all overlapping active transactions – those that are still in

their working phase (rule 1). Rule 2 is automatically fulfilled because the active

transactions do not write until after Tv has completed. Let the active transactions have

(consecutive) transaction identifiers active1 to activeN. The following program describes

the algorithm for the forward validation of Tv:

boolean valid = true;

for (int Tid = active1; Tid <= activeN; Tid++){

if (write set of Tv intersects read set of Tid) valid = false;

}

In Figure 16.28, the write set of transaction Tv must be compared with the read sets of

the transactions with identifiers active1 and active2. (Forward validation should allow

for the fact that read sets of active transactions may change during validation and

writing.) As the read sets of the transaction being validated are not included in the check,

read-only transactions always pass the validation check. As the transactions being

compared with the validating transaction are still active, we have a choice of whether to

abort the validating transaction or to pursue some alternative way of resolving the

conflict. Härder [1984] suggests several alternative strategies:

• Defer the validation until a later time when the conflicting transactions have

finished. However, there is no guarantee that the transaction being validated will

fare any better in the future. There is always the chance that further conflicting

active transactions may start before the validation is achieved.

• Abort all the conflicting active transactions and commit the transaction being

validated.

• Abort the transaction being validated. This is the simplest strategy but has the

disadvantage that future conflicting transactions may be going to abort, in which

case the transaction under validation has aborted unnecessarily.SECTION 16.6 TIMESTAMP ORDERING 727

Comparison of forward and backward validation • We have already seen that forward

validation allows flexibility in the resolution of conflicts, whereas backward validation

allows only one choice – to abort the transaction being validated. In general, the read

sets of transactions are much larger than the write sets. Therefore, backward validation

compares a possibly large read set against the old write sets, whereas forward validation

checks a small write set against the read sets of active transactions. We see that

backward validation has the overhead of storing old write sets until they are no longer

needed. On the other hand, forward validation has to allow for new transactions starting

during the validation process.

Starvation • When a transaction is aborted, it will normally be restarted by the client

program. But in schemes that rely on aborting and restarting transactions, there is no

guarantee that a particular transaction will ever pass the validation checks, for it may

come into conflict with other transactions for the use of objects each time it is restarted.

The prevention of a transaction ever being able to commit is called starvation.

Occurrences of starvation are likely to be rare, but a server that uses optimistic

concurrency control must ensure that a client does not have its transaction aborted

repeatedly. Kung and Robinson suggest that this could be done if the server detects a

transaction that has been aborted several times. They suggest that when the server

detects such a transaction it should be given exclusive access by the use of a critical

section protected by a semaphore.

16.6 Timestamp ordering

In concurrency control schemes based on timestamp ordering, each operation in a

transaction is validated when it is carried out. If the operation cannot be validated, the

transaction is aborted immediately and can then be restarted by the client. Each

transaction is assigned a unique timestamp value when it starts. The timestamp defines

its position in the time sequence of transactions. Requests from transactions can be

totally ordered according to their timestamps. The basic timestamp ordering rule is

based on operation conflicts and is very simple:

A transaction’s request to write an object is valid only if that object was last read and

written by earlier transactions. A transaction’s request to read an object is valid only

if that object was last written by an earlier transaction.

This rule assumes that there is only one version of each object and restricts access to one

transaction at a time. If each transaction has its own tentative version of each object it

accesses, then multiple concurrent transactions can access the same object. The

timestamp ordering rule is refined to ensure that each transaction accesses a consistent

set of versions of the objects. It must also ensure that the tentative versions of each object

are committed in the order determined by the timestamps of the transactions that made

them. This is achieved by transactions waiting, when necessary, for earlier transactions

to complete their writes. The write operations may be performed after the

closeTransaction operation has returned, without making the client wait. But the client

must wait when read operations need to wait for earlier transactions to finish. This728 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

cannot lead to deadlock, since transactions only wait for earlier ones (and no cycle could

occur in the wait-for graph).

Timestamps may be assigned from the server’s clock or, as in the previous section,

a ‘pseudo-time’ may be based on a counter that is incremented whenever a timestamp

value is issued. We defer until Chapter 17 the problem of generating timestamps when

the transaction service is distributed and several servers are involved in a transaction.

We will now describe a form of timestamp-based concurrency control following

the methods adopted in the SDD-1 system [Bernstein et al. 1980] and described by Ceri

and Pelagatti [1985].

As usual, the write operations are recorded in tentative versions of objects and are

invisible to other transactions until a closeTransaction request is issued and the

transaction is committed. Every object has a write timestamp and a set of tentative

versions, each of which has a write timestamp associated with it; each object also has a

set of read timestamps. The write timestamp of the (committed) object is earlier than that

of any of its tentative versions, and the set of read timestamps can be represented by its

maximum member. Whenever a transaction’s write operation on an object is accepted,

the server creates a new tentative version of the object with its write timestamp set to the

transaction timestamp. A transaction’s read operation is directed to the version with the

maximum write timestamp less than the transaction timestamp. Whenever a

transaction’s read operation on an object is accepted, the timestamp of the transaction is

added to its set of read timestamps. When a transaction is committed, the values of the

tentative versions become the values of the objects, and the timestamps of the tentative

versions become the timestamps of the corresponding objects.

In timestamp ordering, each request by a transaction for a read or write operation

on an object is checked to see whether it conforms to the operation conflict rules. A

request by the current transaction Tc can conflict with previous operations done by other

transactions, Ti, whose timestamps indicate that they should be later than Tc. These rules

are shown in Figure 16.29, in which Ti > Tc means Ti is later than Tc and Ti < Tc means

Ti, is earlier than Tc.

Figure 16.29 Operation conflicts for timestamp ordering

Rule Tc Ti

1. write read Tc must not write an object that has been read by any Ti where Ti > Tc.

This requires that Tc  the maximum read timestamp of the object.

2. write write Tc must not write an object that has been written by any Ti where Ti >Tc.

This requires that Tc > the write timestamp of the committed object.

3. read write Tc must not read an object that has been written by any Ti where Ti > Tc.

This requires that Tc > the write timestamp of the committed object.SECTION 16.6 TIMESTAMP ORDERING 729

Timestamp ordering write rule: By combining rules 1 and 2 we get the following rule for

deciding whether to accept a write operation requested by transaction Tc on object D:

if (Tc  maximum read timestamp on D &&

Tc > write timestamp on committed version of D)

perform write operation on tentative version of D with write timestamp Tc

else /\* write is too late \*/

Abort transaction Tc

If a tentative version with write timestamp Tc already exists, the write operation is

addressed to it; otherwise, a new tentative version is created and given write timestamp

Tc. Note that any write that ‘arrives too late’ is aborted – it is too late in the sense that a

transaction with a later timestamp has already read or written the object.

Figure 16.30 illustrates the action of a write operation by transaction T3 in cases

where T3  maximum read timestamp on the object (the read timestamps are not

shown). In cases (a) to (c), T3 > write timestamp on the committed version of the object

and a tentative version with write timestamp T3 is inserted at the appropriate place in the

list of tentative versions ordered by their transaction timestamps. In case (d),

T3 < write timestamp on the committed version of the object and the transaction is

aborted.

Figure 16.30 Write operations and timestamps

(a) T3 write (b) T3 write

(c) T3 write (d) T3 write

object produced by transaction Ti

(with write timestamp Ti)

T1 < T2 < T3 < T4

Time

Before

After

T2 T2

T3

Time

Before

After

T2 T2

T3

T1 T1

Time

Before

After

T1 T1

T4 T3

T4

Time

Transaction

aborts

Before

After

T4 T4

Key:

Committed Tentative

Ti Ti730 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

Timestamp ordering read rule: By using rule 3 we arrive at the following rule for

deciding whether to accept immediately, to wait or to reject a read operation requested

by transaction Tc on object D:

if ( Tc > write timestamp on committed version of D) {

let Dselected be the version of D with the maximum write timestamp ð Tc

if (Dselected is committed)

perform read operation on the version Dselected

else

wait until the transaction that made version Dselected commits or aborts

then reapply the read rule

} else

Abort transaction Tc

Note:

• If transaction Tc has already written its own version of the object, this will be used.

• A read operation that arrives too early waits for the earlier transaction to complete.

If the earlier transaction commits, then Tc will read from its committed version. If

it aborts, then Tc will repeat the read rule (and select the previous version). This

rule prevents dirty reads.

• A read operation that ‘arrives too late’ is aborted – it is too late in the sense that a

transaction with a later timestamp has already written the object.

Figure 16.31 illustrates the timestamp ordering read rule. It includes four cases labelled

(a) to (d), each of which illustrates the action of a read operation by transaction T3. In

each case, a version whose write timestamp is less than or equal to T3 is selected. If such

a version exists, it is indicated with a line. In cases (a) and (b) the read operation is

directed to a committed version – in (a) it is the only version, whereas in (b) there is a

tentative version belonging to a later transaction. In case (c) the read operation is

directed to a tentative version and must wait until the transaction that made it commits

or aborts. In case (d) there is no suitable version to read and transaction T3 is aborted.

When a coordinator receives a request to commit a transaction, it will always be

able to do so because all the operations of transactions are checked for consistency with

those of earlier transactions before being carried out. The committed versions of each

object must be created in timestamp order. Therefore, a coordinator sometimes needs to

wait for earlier transactions to complete before writing all the committed versions of the

objects accessed by a particular transaction, but there is no need for the client to wait. In

order to make a transaction recoverable after a server crash, the tentative versions of

objects and the fact that the transaction has committed must be written to permanent

storage before acknowledging the client’s request to commit the transaction.

Note that this timestamp ordering algorithm is a strict one – it ensures strict

executions of transactions (see Section 16.2). The timestamp ordering read rule delays

a transaction’s read operation on any object until all transactions that had previously

written that object have committed or aborted. The arrangement to commit versions in

order ensures that the execution of a transaction’s write operation on any object is

delayed until all transactions that had previously written that object have committed or

aborted.SECTION 16.6 TIMESTAMP ORDERING 731

In Figure 16.32, we return to our illustration concerning the two concurrent

banking transactions T and U introduced in Figure 16.7. The columns headed A, B and

C refer to information about accounts with those names. Each account has an entry RTS

that records the maximum read timestamp and an entry WTS that records the write

timestamp of each version – with timestamps of committed versions in bold. Initially,

all accounts have committed versions written by transaction S, and the set of read

timestamps is empty. We assume S < T < U. The example shows that when transaction

U is ready to get the balance of B it will wait for T to complete so that it can read the

value set by T if it commits.

The timestamp method just described does avoid deadlocks, but it is quite likely

to cause restarts. A modification known as the ‘ignore obsolete write’ rule is an

improvement. This is a modification to the timestamp ordering write rule:

If a write is too late it can be ignored instead of aborting the transaction, because if it

had arrived in time its effects would have been overwritten anyway. However, if

another transaction has read the object, the transaction with the late write fails due to

the read timestamp on the item.

Multiversion timestamp ordering • In this section, we have shown how the

concurrency provided by basic timestamp ordering is improved by allowing each

transaction to write its own tentative versions of objects. In multiversion timestamp

ordering, which was introduced by Reed [1983], a list of old committed versions as well

as tentative versions is kept for each object. This list represents the history of the values

Figure 16.31 Read operations and timestamps

(a) T3 read (b) T3 read

(c) T3 read (d) T3 read

object produced by transaction Ti

(with write timestamp Ti)

T1 < T2 < T3 < T4

Time

read

proceeds

Selected

T2

Time

read

proceeds

Selected

T2 T4

Time

read waits

Selected

T1 T2

Time

Transaction

T4 aborts

Key:

Committed Tentative

Ti Ti732 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

of the object. The benefit of using multiple versions is that read operations that arrive

too late need not be rejected.

Each version has a read timestamp recording the largest timestamp of any

transaction that has read from it in addition to a write timestamp. As before, whenever

a write operation is accepted, it is directed to a tentative version with the write timestamp

of the transaction. Whenever a read operation is carried out, it is directed to the version

with the largest write timestamp less than the transaction timestamp. If the transaction

timestamp is larger than the read timestamp of the version being used, the read

timestamp of the version is set to the transaction timestamp.

When a read arrives late, it can be allowed to read from an old committed version,

so there is no need to abort late read operations. In multiversion timestamp ordering,

read operations are always permitted, although they may have to wait for earlier

transactions to complete (either commit or abort), which ensures that executions are

recoverable. See Exercise 16.22 for a discussion of the possibility of cascading aborts.

This deals with rule 3 in the conflict rules for timestamp ordering.

There is no conflict between write operations of different transactions, because

each transaction writes its own committed version of the objects it accesses. This

removes rule 2 in the conflict rules for timestamp ordering, leaving us with:

Rule 1. Tc must not write objects that have been read by any Ti where Ti > Tc.

This rule will be broken if there is any version of the object with read timestamp > Tc,

but only if this version has a write timestamp less than or equal to Tc. (This write cannot

have any effect on later versions.)

Figure 16.32 Timestamps in transactions T and U

Timestamps and versions of objects

T U A B C

RTS WTS RTS WTS RTS WTS

{} S {} S {} S

openTransaction

bal = b.getBalance() {T}

openTransaction

b.setBalance(bal\*1.1) S, T

bal = b.getBalance()

wait for T

a.withdraw(bal/10) ••• S, T

commit ••• T T

bal = b.getBalance() {U}

b.setBalance(bal\*1.1) T, U

c.withdraw(bal/10) S, USECTION 16.6 TIMESTAMP ORDERING 733

Multiversion timestamp ordering write rule: As any potentially conflicting read operation

will have been directed to the most recent version of an object, the server inspects the

version DmaxEarlier with the maximum write timestamp less than or equal to Tc. We have

the following rule for performing a write operation requested by transaction Tc on object

D:

if (read timestamp of DmaxEarlier  Tc)

perform write operation on a tentative version of D with write timestamp Tc

else abort transaction Tc

Figure 16.33 illustrates an example where a write is rejected. The object already has

committed versions with write timestamps T1 and T2. The object receives the following

sequence of requests for operations on the object:

T3 read; T3 write; T5 read; T4 write.

1. T3 requests a read operation, which puts a read timestamp T3 on T2’s version.

2. T3 requests a write operation, which makes a new tentative version with write

timestamp T3.

3. T5 requests a read operation, which uses the version with write timestamp T3 (the

highest timestamp that is less than T5).

4. T4 requests a write operation, which is rejected because the read timestamp T5 of

the version with write timestamp T3 is bigger than T4. (If it were permitted, the

write timestamp of the new version would be T4. If such a version were allowed,

then it would invalidate T5’s read operation, which should have used the version

with timestamp T4.)

Figure 16.33 Late write operation would invalidate a read

object produced by transaction

Ti (with write timestamp Ti and

read timestamp Tk)

Time

T T4 write;

T3 read; T3 write; 5 read;

T2

T1 T3 T T5 3

T1 < T2 < T3 < T4 < T5

Key:

Committed Tentative

Ti Ti

Tk Tk734 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

When a transaction is aborted, all the versions that it created are deleted. When a

transaction is committed, all the versions that it created are retained, but to control the

use of storage space, old versions must be deleted from time to time. Although it has the

overhead of storage space, multiversion timestamp ordering does allow considerable

concurrency, does not suffer from deadlocks and always permits read operations. For

further information about multiversion timestamp ordering, see Bernstein et al. [1987].

16.7 Comparison of methods for concurrency control

We have described three separate methods for controlling concurrent access to shared

data: strict two-phase locking, optimistic methods and timestamp ordering. All of the

methods carry some overheads in the time and space they require, and they all limit to

some extent the potential for concurrent operation.

The timestamp ordering method is similar to two-phase locking in that both use

pessimistic approaches in which conflicts between transactions are detected as each

object is accessed. On the one hand, timestamp ordering decides the serialization order

statically – when a transaction starts. On the other hand, two-phase locking decides the

serialization order dynamically – according to the order in which objects are accessed.

Timestamp ordering, and in particular multiversion timestamp ordering, is better than

strict two-phase locking for read-only transactions. Two-phase locking is better when

the operations in transactions are predominantly updates.

Some work uses the observation that timestamp ordering is beneficial for

transactions with predominantly read operations and that locking is beneficial for

transactions with more writes than reads as an argument for allowing hybrid schemes in

which some transactions use timestamp ordering and others use locking for concurrency

control. Readers who are interested in the use of mixed methods should read Bernstein

et al. [1987].

The pessimistic methods differ in the strategy used when a conflicting access to

an object is detected. Timestamp ordering aborts the transaction immediately, whereas

locking makes the transaction wait – but with a possible later penalty of aborting to

avoid deadlock.

When optimistic concurrency control is used, all transactions are allowed to

proceed, but some are aborted when they attempt to commit, or in forward validation

transactions are aborted earlier. This results in relatively efficient operation when there

are few conflicts, but a substantial amount of work may have to be repeated when a

transaction is aborted.

Locking has been in use for many years in database systems, but timestamp

ordering has been used in the SDD-1 database system. Both methods have been used in

file servers. However, historically, the predominant method of concurrency control of

access to data in distributed systems is by locking – for example, as mentioned earlier,

the CORBA Concurrency Control Service is based entirely on the use of locks. In

particular, it provides hierarchic locking, which allows for mixed-granularity locking on

hierarchically structured data.SECTION 16.7 COMPARISON OF METHODS FOR CONCURRENCY CONTROL 735

Several research distributed systems, for example Argus [Liskov 1988] and

Arjuna [Shrivastava et al. 1991], have explored the use of semantic locks, timestamp

ordering and new approaches to long transactions.

Ellis et al. [1991] wrote a review of requirements for multi-user applications in

which all users expect to see common views of objects being updated by any of the

users. Many of the schemes provided notification of changes made by other users, but

this is contrary to the idea of isolation.

Barghouti and Kaiser [1991] wrote a review of what are sometimes described as

‘advanced database applications’ – for example, cooperative CAD/CAM and software

development systems. In such applications, transactions last for a long time, and users

work on independent versions of objects that are checked out from a common database

and checked in when the work is finished. The merging of versions requires cooperation

between users.

Simililarly, the above concurrency control mechanisms are not always adequate

for twenty-first-century applications that enable users to share documents over the

Internet. Many of the latter use optimistic forms of concurrency control followed by

conflict resolution instead of aborting one of any pair of conflicting operations.

The following are some examples.

Dropbox • Dropbox [www.dropbox.com] is a cloud service that provides file backup

and enables users to share files and folders, accessing them from anywhere. Dropbox

uses an optimistic form of concurrency control, keeping track of consistency and

preventing clashes between users’ updates – which are at the granularity of whole files.

Thus if two users make concurrent updates to the same file, the first write will be

accepted and the second rejected. However, Dropbox provides a version history to

enable users to merge their updates manually or restore previous versions.

Google apps • Google Apps [www.google.com I] are listed in Figure 21.2. They

include Google Docs, a cloud service that provides web-based applications (word

processor, spreadsheet and presentation) that allow users to collaborate with one

another by means of shared documents. If several people edit the same document

simultaneously, they will see each other’s changes. In the case of a word processor

document, users can see one another’s cursors and updates are shown at the level of

individual characters as they are typed by any participant. Users are left to resolve any

conflicts that occur, but conflicts are generally avoided because users are continuously

aware of each other’s activities. In the case of a spreadsheet document, users’ cursors

and changes are displayed and updated at the granularity of single cells. If two users

access the same cell simultaneously, the last update wins.

Wikipedia • Concurrency control for editing is optimistic, allowing editors concurrent

access to web pages in which the first write is accepted and a user making a subsequent

write is shown an ‘edit conflict’ screen and asked to resolve the conflicts.

Dynamo • Amazon.com’s key-value storage service uses optimistic concurrency

control with conflict resolution (see the box on the next page).736 CHAPTER 16 TRANSACTIONS AND CONCURRENCY CONTROL

16.8 Summary

Transactions provide a means by which clients can specify sequences of operations that

are atomic in the presence of other concurrent transactions and server crashes. The first

aspect of atomicity is achieved by running transactions so that their effects are serially

equivalent. The effects of committed transactions are recorded in permanent storage so

that the transaction service can recover from process crashes. To allow transactions the

ability to abort, without having harmful side effects on other transactions, executions

must be strict – that is, reads and writes of one transaction must be delayed until other

transactions that wrote the same objects have either committed or aborted. To allow

transactions the choice of either committing or aborting, their operations are performed

in tentative versions that cannot be accessed by other transactions. The tentative

versions of objects are copied to the real objects and to permanent storage when a

transaction commits.

Nested transactions are formed by structuring transactions from other subtransactions. Nesting is particularly useful in distributed systems because it allows

concurrent execution of subtransactions in separate servers. Nesting also has the

advantage of allowing independent recovery of parts of a transaction.

Operation conflicts form a basis for the derivation of concurrency control

protocols. Protocols must not only ensure serializability but also allow for recovery by

using strict executions to avoid problems associated with transactions aborting, such as

cascading aborts.

Three alternative strategies are possible in scheduling an operation in a

transaction. They are (1) to execute it immediately, (2) to delay it or (3) to abort it.