

# DISTRIBUTED SYSTEMS

## Lecture notes

### IMPORTANT NOTES:

1. *Only for the master students of Computer Science Department of UVT*
2. *Posting this material on a public sites outside UVT is prohibit due to the copyright of the books that are referred*
3. *This material is only a text compilation from the cited books on distributed systems intended to easy their access by the UVT students*

# Content

## PART I. PRINCIPLES OF DISTRIBUTED SYSTEMS

1.1 What Is a Distributed System? .....	1
1.1.1 Goals.....	1
1.1.1.1 Advantages of Distributed Systems over Centralized Systems.....	2
1.1.1.2 Advantages of Distributed Systems over Independent PCs .....	3
1.1.1.3 Advantages of Distributed Computing Environment over Standalone Application .....	3
1.1.1.4 Disadvantages of Distributed Systems.....	4
1.1.2 Software Concepts .....	4
1.1.3 Design Issues .....	5
1.1.3.1 Transparency.....	5
1.1.3.2 Flexibility.....	7
1.1.3.3 Reliability .....	7
1.1.3.4 Performance .....	8
1.1.3.5 Scalability .....	9
1.1.4 Middleware .....	9
1.1.4.1 Middleware Tasks .....	10
1.1.4.2 The Structure of a Middleware Platform .....	10
1.1.4.3 Standardization of a Middleware .....	11
1.1.4.4 Portability and Interoperability .....	11
1.1.5 Sample Application .....	12
1.1.6 Problems .....	13
1.2 Communication in Distributed Systems .....	13
1.2.1 The Client-Server Model .....	15
1.2.1.1 Addressing .....	15
1.2.1.2 Blocking versus Nonblocking Primitives.....	16
1.2.1.3 Implementing the Client-Server Model .....	18
1.2.2 Remote Procedure Call (RPC).....	18
1.2.2.1 Basic RPC Operation .....	18
1.2.2.2 Parameter Passing .....	20
1.2.2.3 Dynamic Binding .....	22
1.2.2.4 RPC Protocols.....	23
1.2.2.5 Failure Semantics.....	24
1.2.3 Group Communication .....	25
1.2.4 Problems .....	27
1.3 Synchronization in Distributed Systems.....	28
1.3.1 Clock Synchronization .....	28
1.3.1.1 Logical Clocks .....	29
1.3.1.2 Physical Clocks.....	31
1.3.1.3 Clock Synchronization Algorithms.....	32
1.3.1.4 Multiple External Time Sources .....	35
1.3.1.5 Use of Synchronized Clocks .....	35
1.3.2 Mutual Exclusion.....	35
1.3.2.1 A Centralized Algorithm.....	36
1.3.2.2 A Distributed Algorithm .....	37
1.3.2.3 A Token Ring Algorithm .....	38
1.3.2.4 Comparison of the Three Algorithms .....	39
1.3.3 Election Algorithms.....	39
1.3.3.1 The Bully Algorithm.....	39
1.3.3.2 Ring Algorithm .....	40
1.3.4 Atomic Transactions.....	41
1.3.4.1 The Transaction Model .....	41
1.3.4.2 Transaction Primitives .....	42
1.3.4.3 Implementation .....	43
1.3.4.4 Concurrency Control.....	45
1.3.5 Deadlocks in Distributed Systems .....	48
1.3.5.1 Distributed Deadlock Detection.....	48
1.3.6 Problems .....	51
1.4 System Models .....	52
1.4.1 The Workstation Model.....	52
1.4.2 Using Idle Workstations .....	52

1.4.3 The Processor Pool Model .....	54
1.4.4 Hybrid Model .....	55
1.4.5 Fault Tolerance .....	56
1.4.5.1 Component Faults .....	56
1.4.5.2 System Failures .....	57
1.4.5.3 Synchronous versus Asynchronous Systems .....	57
1.4.5.4 Use of Redundancy .....	58
1.4.5.5 Fault Tolerance Using Active Replication .....	58
1.4.5.6 Fault Tolerance Using Primary Backup .....	59
1.4.5.7 Agreement in Faulty Systems .....	61
1.4.6 Real-Time Distributed Systems .....	62
1.4.6.1 What Is a Real-Time System? .....	62
1.4.6.2 Design Issues .....	64
1.4.6.3 Real-Time Communication .....	65
1.4.7 Problems .....	67

## **PART II. ARCHITECTURES OF MODERN DISTRIBUTED SYSTEMS**

2.1 Peer-To-Peer Architecture .....	69
2.1.1 A Brief History of P2P .....	69
2.1.2 Definition and Motivation .....	69
2.1.2.1 Peer-To-Peer Application .....	70
2.1.2.2 Motivation to Adopt P2P .....	70
2.1.3 Functionality .....	71
2.1.3.1 P2P Architectural Details .....	71
2.1.3.2 How P2P Forms Dynamic Networks .....	72
2.1.3.3 Discovery .....	72
2.1.3.4 Broadcast .....	73
2.1.3.5 Resource Indexing .....	74
2.1.3.6 Node Autonomy .....	74
2.1.3.7 Peer of Equals .....	74
2.1.3.8 Supporting Mixed Models .....	75
2.1.4 The Future Includes Web Services .....	77
2.1.5 P2P Application Types .....	76
2.1.5.1 Instant Messaging .....	76
2.1.5.2 Managing and Sharing Information .....	78
2.1.5.3 Collaboration .....	78
2.1.5.4 Distributed Services .....	79
2.1.6 JXTA .....	80
2.2 Grids .....	81
2.2.1 Computational Grids .....	81
2.2.2 Service Grids .....	83
2.3 Service-Oriented Architecture .....	84
2.3.1 SOA – an Architectural Style .....	84
2.3.2 Service Abstraction .....	85
2.3.3 SOA Objectives .....	86
2.3.4 Advantages of SOA .....	87
2.3.5 Technologies used to implement SOA .....	87
2.3.6 SOA using Web Services .....	91
2.4 Ubiquitous computing .....	92
2.4.1 Definition .....	92
2.4.2 Middleware Challenges .....	92
2.4.2.1 Constrained Resources .....	92
2.4.2.2 Network Dynamics .....	93
2.4.2.3 Scale of Deployment .....	94
2.4.2.4 Real-world Integration .....	94
2.4.2.5 Collection, Processing and Storage of Sensory Data .....	94
2.4.2.6 Integration with Background Infrastructures .....	95
2.4.3 Case Study: Sensor Networks .....	95
2.4.3.1 Databases .....	95
2.4.3.2 Mobile agents .....	96
2.4.3.3 Events .....	97
2.5 Virtualization .....	98
2.5.1 What Is Virtualization? .....	98

2.5.1.1 Application Virtualization .....	98
2.5.1.2 Desktop Virtualization .....	99
2.5.1.3 Network Virtualization .....	99
2.5.1.4 Server and Machine Virtualization .....	99
2.5.1.5 Storage Virtualization .....	101
2.5.1.6 System - Level or Operating System Virtualization .....	102
2.5.1.7 Why Virtualization Today?.....	103
2.5.1.8 Advantages of Virtualization .....	103
2.5.1.9 Disadvantages of Virtualization.....	105
2.5.2 Basic Approaches to Virtual Systems.....	105
2.6 Cloud computing .....	106
2.6.1 Software as a Service.....	108
2.6.2 Platform as a Service.....	108
2.6.3 Infrastructure as a Service .....	108
2.6.4 Anything as a Service .....	109
2.6.5 Pioneers.....	110

### **PART III. DISTRIBUTED PROGRAMMING IN JAVA**

3.1 Modern Technologies for Distributed Applications .....	115
3.1.1 CORBA .....	115
3.1.2 Java RMI .....	116
3.1.3 Microsoft DCOM .....	118
3.1.4 Message-Oriented Middleware (MOM) .....	119
3.1.5 Common Challenges in Distributed Computing .....	119
3.1.6 The Role of J2EE and XML in Distributed Computing .....	120
3.1.7 Web Services .....	121
3.2 Sockets in Java .....	122
3.2.1 Networks, Packets, and Protocols.....	122
3.2.1.1 About Addresses .....	123
3.2.1.2 About Names .....	124
3.2.1.3 Clients, Servers and Peers.....	125
3.2.1.4 Sockets and Ports .....	126
3.2.1.5 Standard Services.....	127
3.2.1.6 URLs and DNS .....	128
3.2.2 Basic Sockets .....	128
3.2.2.1 Socket Addresses .....	132
3.2.2.2 Fast overview of Java classes for TCP and UDP sockets .....	133
3.2.2.3 Multicast Sockets .....	134
3.2.3 TCP Sockets .....	135
3.2.3.1 TCP Sockets in Java .....	136
3.2.3.2 Example of TCP Client .....	134
3.2.3.3 Example of TCP Server .....	138
3.2.3.4 Another Example .....	140
3.2.4 Streams, Readers, and Writers for Input and Output .....	144
3.2.4.1 Input and Output Streams .....	144
3.2.4.2 Readers and Writers .....	144
3.2.4.3 Another Example .....	149
3.2.4.4 Security .....	154
3.2.5 UDP Sockets.....	156
3.2.5.1 DatagramPacket .....	156
3.2.5.2 Example of UDP Client .....	157
3.2.5.3 Example of UDP Server.....	159
3.2.5.4 Another Example .....	161
3.2.5.5 Sending and Receiving with UDP Sockets .....	166
3.2.5.6 Exercises .....	167
3.2.6 Sending and Receiving Encoded Data.....	168
3.2.6.1 Encoding Information .....	168
3.2.6.2 Composing I/O Streams .....	173
3.2.6.3 Framing and Parsing .....	173
3.2.6.4 Java-Specific Encodings .....	177
3.2.6.5 Constructing and Parsing Protocol Messages .....	177
3.2.6.6 Sending and Receiving over the stream .....	182
3.2.6.7 Exercises .....	186
3.2.7 Multitasking.....	187

3.2.7.1 Mutithreaded server .....	187
3.2.7.2 Thread-per-Client.....	189
3.2.7.3 Thread Pool.....	190
3.2.7.4 System-Managed Dispatching: The Executor Interface.....	192
3.2.7.5 Blocking and Timeouts .....	193
3.2.7.6 Multiple Recipients.....	195
3.2.7.7 Exercises .....	198
3.2.8 Distributing Objects.....	199
3.2.8.1 Why Distribute Objects?.....	199
3.2.8.2 Creating Remote Objects .....	199
3.2.8.3 Remote Method Calls .....	200
3.2.8.4 Other Issues.....	201
3.2.8.5 Features of Distributed Object Systems .....	202
3.2.8.6 Distributed Object Schemes for Java .....	204
3.2.9 Remote Method Invocation (RMI) .....	206
3.2.9.1 The Basic RMI Process.....	206
3.2.9.2 Java RMI.....	206
3.2.9.3 Implementation Details .....	209
3.2.9.4 Using RMI Meaningfully.....	211
3.2.9.5 Example of a RMI Solver .....	216
3.2.9.6 RMI Security.....	220
3.2.9.7 Exercises .....	221
3.2.10 URLConnections and ClassLoader.....	222
3.2.10.1 When and Where Are URLs Practical? .....	223
3.2.10.2 The ClassLoader .....	224
3.2.10.3 Loading Classes from the Network.....	224
3.2.11 Network Programming with GUIs.....	228
3.2.12 Downloading Web Pages.....	232
3.2.13 NIO – New I/O packages.....	234
3.2.13.1 NIO advantages.....	234
3.2.13.2 Using Channels with Buffers .....	235
3.2.13.3 Selectors.....	236
3.2.13.4 Buffers in Detail.....	240
3.2.13.5 Exercises .....	252
3.3 Web Services.....	253
3.3.1 Web Services Architectures.....	253
3.3.1.1 Definitions of Web Services .....	253
3.3.1.2 Historical Evolution .....	254
3.3.1.3 Web Services Architecture as a Distributed Computing Architecture .....	255
3.3.1.4 Another Way of Defining What Web Services Do: Publish, Find, and Bind .....	255
3.3.1.5 Types of Web Service Architectures .....	256
3.3.1.6 Elements of the Web Services Platform.....	257
3.3.1.7 Benefits of Web Services.....	258
3.3.1.8 Defining Objects and Web Services .....	259
3.3.1.9 Web Services Communication Models .....	261
3.3.1.10 Overview of Web Service Standards .....	261
3.3.1.11 Web Services Protocol Stack .....	262
3.3.1.12 Web Service Implementations .....	265
3.3.2 XML .....	267
3.3.2.1 Overview of XML .....	267
3.3.2.2 Building an XML Document .....	268
3.3.2.3 XML—Advantages and Disadvantages .....	275
3.3.3 WSDL.....	277
3.3.3.1 Service Contracts .....	277
3.3.3.2 WSDL in the Context of Remote Object Technologies .....	278
3.3.3.3 Structure of the WSDL Document.....	280
3.3.4 SOAP .....	292
3.3.4.1 SOAP Concept .....	292
3.3.4.2 SOAP Vocabulary.....	294
3.3.4.3 A Short Description .....	296
3.3.4.4 Basic SOAP Document .....	299
3.3.4.5 SOAP Data Types and Structures .....	305
3.3.4.6 SOAP Message Exchange Model .....	310
3.3.4.7 SOAP Communication.....	311
3.3.4.8 SOAP Documents and Transport Bindings.....	313

3.3.4.9 SOAP Security .....	321
3.3.5 UDDI .....	325
3.3.5.1 Service Lookup through UDDI.....	325
3.3.5.2 Registry Standards .....	325
3.3.5.2 UDDI Overview .....	326
3.3.5.3 UDDI Case Studies .....	328
3.3.5.4 UDDI and XML .....	328
3.3.6 REST .....	330
3.3.6.1 Key ideas of REST .....	330
3.3.6.2 Key components .....	331
3.3.6.3 REST Style Web Services .....	331
3.3.7 Other Industry Standards Supporting Web Services.....	332
3.3.8 Web Services with Apache SOAP.....	334
3.3.8.1 Creating and Deploying an Apache SOAP Web Service.....	334
3.3.8.2 Creating Web Service Consumers in Java, Java Servlets, and JSP Pages .....	338
3.3.9 Web Services with Apache Axis .....	347
3.3.9.1 Creating and Deploying an Axis WS: Differences between Apache Axis and SOAP.....	347
3.3.9.2 Creating a Consumer with Axis .....	349
3.3.9.3 A Hello Example .....	355
3.3.9.4 Overview of Axis2 .....	359
3.3.10 Web Services Support in J2EE .....	360
3.3.10.1 Platform Overview.....	360
3.3.10.2 APIs .....	361
3.3.10.3 Java APIs for XML Processing (JAXP).....	362
3.3.10.4 Java API for XML-Based RPC (JAX-RPC) .....	363
3.3.10.5 Java API for XML Registries (JAXR) .....	365
3.3.10.6 SOAP with Attachments API for Java (SAAJ) .....	365
3.3.10.7 Web Service Technologies Integrated in J2EE Platform .....	366
3.3.10.8 Flow of a Web Service Call .....	370
3.3.10.9 Designing the Interface .....	371
3.3.10.10 Receiving Requests .....	379
3.3.10.11 Publishing a Web Service .....	383
3.3.10.12 Handling XML Documents in a Web Service .....	385
3.3.10.13 Deploying and Packaging a Service Endpoint .....	388
3.3.10.14 Client Design .....	391
3.3.10.15 Developing Client Applications to Use a Web Service .....	396
3.3.10.16 Java Web Services Developer Pack (JWSDP) .....	409
3.3.10.17 Other Java-XML Technologies.....	409
3.3.10.18 JAXB and XML Over HTTP Protocol.....	410
3.3.10.19 JAX-WS 2 .....	419
3.3.11 JXTA .....	430
3.3.11.1 The JXTA Protocols .....	430
3.3.11.2 JXTA Services .....	435
3.3.11.3 JXTA Modules .....	436
3.3.11.4 The JXTA J2SE API.....	437
3.3.11.5 First JXTA Program.....	437
3.3.11.6 JXTA Prime Cruncher .....	436
3.3.11.7 P2P Web Services .....	450
3.3.12 Web Services Using Spring .....	461
3.3.12.1 Overview of Spring-WS .....	466
3.3.12.2 A Hello example .....	466
3.3.13 Web Services Using XFire .....	464
3.3.14 Data and Services .....	467
3.3.14.1 Java Data Objects (JDO).....	463
3.3.14.2 JDO Using JPOX .....	466
3.3.14.3 Data Services .....	469
3.3.14.4 Service Data Objects (SDO) .....	470
3.3.14.5 SDO Architecture .....	470
3.3.14.6 Using Apache Tuscany SDO .....	471
3.3.15 Service Component Architecture.....	475
3.3.15.1 SCA concept .....	475
3.3.15.2 Using Apache Tuscany SCA Java .....	476
3.3.16 WS-BPEL .....	478
3.3.16.1 Orchestration.....	478
3.3.16.2 Choreography .....	479

3.3.16.3 WS-BPEL language basics .....	481
3.3.17 Web Services with Eclipse WTP .....	487
3.3.17.1 Iteration 1: Developing Web Services Top-Down .....	487
3.3.17.2 Iteration 2: Developing Web Services Bottom-Up .....	493
3.3.17.3 Iteration 3: Generating Web Service Client Proxies .....	495
3.3.17.4 Iteration 4: Testing Web Services for Interoperability .....	497
3.3.17.5 Iteration 5: Using Web Services in Web Applications.....	500
3.3.17.6 Iteration 6: Discovering and Publishing Web Services.....	502
3.3.18 Web Services Support in .NET .....	507
3.3.19 Web Services with Ajax and PHP .....	510
3.3.19.1 Reinventing the Web.....	510
3.3.19.2 Ajax and Web Services .....	513
3.3.20 Web 2.0 Technologies and Web Services.....	518
3.3.20.1 Syndication .....	518
3.3.20.2 Mashups .....	521
3.3.21 Publicly Available Web Services .....	525
3.3.21.1 Blogging Services .....	525
3.3.21.2 Bookmark Services .....	527
3.3.21.3 Financial Services .....	528
3.3.21.4 Mapping Services .....	529
3.3.21.5 Music/Video Services .....	532
3.3.21.6 News/Weather Services .....	534
3.3.21.7 Photo Services.....	537
3.3.21.8 Reference Services.....	539
3.3.21.9 Search Services .....	540
3.3.21.10 Shopping Services.....	541
3.3.21.11 Other Services.....	542
3.3.22 Amazon Web Services.....	544
3.3.22.1 Infrastructure in the Cloud .....	544
3.3.22.2 S3: Simple Storage Service .....	548
3.3.22.3 EC2: Elastic Compute Cloud .....	553
3.3.23 Google Web Services .....	556
3.3.23 Searching services.....	556
3.3.24 Programming with Google App Engine .....	560

# PART I.

## PRINCIPLES OF DISTRIBUTED SYSTEMS

## 1.1 What Is a Distributed System?

Various definitions of distributed systems have been given in the literature, none of them satisfactory and none of them in agreement with any of the others. We enumerate here some of them.

*A distributed system is a collection of independent computers that appear to the users of the system as a single computer (Tanenbaum, 1994).*

This definition has two aspects. The first one deals with hardware: the machines are autonomous. The second one deals with software: the users think of the system as a single computer. Both are essential.

Rather than going further with definitions, it is probably more helpful to give several examples of distributed systems. As a first example, consider a network of workstations in a university or company department. In addition to each user's personal workstation, there might be a pool of processors in the machine room that are not assigned to specific users but are allocated dynamically as needed. Such a system might have a single file system, with all files accessible from all machines in the same way and using the same path name. Furthermore, when a user typed a command, the system could look for the best place to execute that command, possibly on the user's own workstation, possibly on an idle workstation belonging to someone else, and possibly on one of the unassigned processors in the machine room. If the system as a whole looked and acted like a classical single-processor timesharing system, it would qualify as a distributed system.

As a second example, consider a factory full of robots, each containing a powerful computer for handling vision, planning, communication, and other tasks. When a robot on the assembly line notices that a part it is supposed to install is defective, it asks another robot in the parts department to bring it a replacement. If all the robots act like peripheral devices attached to the same central computer and the system can be programmed that way, it too counts as a distributed system.

As a final example, think about a large bank with hundreds of branch offices all over the world. Each office has a master computer to store local accounts and handle local transactions. In addition, each computer has the ability to talk to all other branch computers and with a central computer at headquarters. If transactions can be done without regard to where a customer or account is, and the users do not notice any difference, it too would be considered a distributed system.

*A distributed system is an information-processing system that contains a number of independent computers that cooperate with one another over a communications network in order to achieve a specific objective (modern definition).*

This definition pinpoints a number of aspects of distributed systems. Although the elementary unit of a distributed system is a computer that is networked with other computers, the computer is autonomous in the way it carries out its actions. Computers are linked to one another over a communications network that enables an exchange of messages between computers. The objective of this message exchange is to achieve a cooperation between computers for the purpose of attaining a common goal.

A physical view of a distributed system includes computers as nodes of the communications network along with details about the communications network itself. In contrast, a logical view of a distributed system highlights the applications aspects: a distributed system can also be interpreted as a set of cooperating processes. The distribution aspect refers to the distribution of state (data) and behavior (code) of an application. The process encapsulates part of the state and part of the behavior of an application, and the application's semantics are achieved through the cooperation of several processes. The logical distribution is independent of the physical one. For example, processes do not necessarily have to be linked over a network but instead can all be found on one computer.

### 1.1.1 Goals

In this section we will discuss the motivation and goals of typical distributed systems and look at their advantages and disadvantages compared to traditional centralized systems.

### 1.1.1.1 Advantages of Distributed Systems over Centralized Systems

Distributed systems offer a variety of advantages compared to centrally organized servers. Decentralization is a more economic option because networked computing systems offer a better price/performance ratio than server systems. The introduction of redundancy increases availability when parts of a system fail. Applications that can easily be run simultaneously also offer benefits in terms of faster performance vis-à-vis centralized solutions. Distributed systems can be extended through the addition of components, thereby providing better scalability compared to centralized systems.

The real driving force behind the trend toward decentralization is economics. The most cost-effective solution is frequently to harness a large number of cheap CPUs together in a system. Thus the leading reason for the trend toward distributed systems is that these systems potentially have a much better price/performance ratio than a single large centralized system would have. In effect, a distributed system gives more bang for the buck.

A slight variation on this theme is the observation that a collection of microprocessors cannot only give a better ratio price/performance than a high performance computing system. Einstein's theory of relativity dictates that nothing can travel faster than light, which can cover only 0.6 mm in 2 picosec. Practically, a computer of that speed fully contained in a 0.6-mm cube would generate so much heat that it would melt instantly. Thus whether the goal is normal performance at low cost or extremely high performance at greater cost, distributed systems have much to offer. Usually one makes a distinction between distributed systems, which are designed to allow many users to work together, and parallel systems, whose only goal is to achieve maximum speedup on a single problem. This distinction is difficult to maintain because the design spectrum is really a continuum (see Figure 1.1). We prefer to use the term "distributed system" in the broadest sense to denote any system in which multiple interconnected CPUs work together.

## The Computing Continuum



*Fig. 1.1. The computing continuum*

A next reason for building a distributed system is that some applications are inherently distributed. A supermarket chain might have many stores, each of which gets goods delivered locally (possibly from local farms), makes local sales, and makes local decisions about which vegetables are so old or rotten that they must be thrown out. It therefore makes sense to keep track of inventory at each store on a local computer rather than centrally at corporate headquarters. After all, most queries and updates will be done locally. Nevertheless, from time to time, top management may want to find out how many rutabagas it currently owns. One way to accomplish this goal is to make the complete system look like a single computer to the application programs, but implement decentrally, with one computer per store as we have described. This would then be a commercial distributed system.

Another inherently distributed system is what is often called supported cooperative work, in which a group of people, located far from each other, are working together, for example, to produce a joint report. Another example: computer-supported cooperative games, in which players at different locations play against each other in real time.

Another potential advantage of a distributed system over a centralized system is higher reliability. By distributing the workload over many machines, a single chip failure will bring down at most one machine, leaving the rest intact. Ideally, if a very small percent of the machines are down at any moment, the system should be able to continue to work with a percent loss in performance. For critical applications, such as control of nuclear reactors or aircraft, using a distributed system to achieve high reliability may be the dominant consideration.

Finally, incremental growth is also potentially a big plus. Often, a company will buy a server with the intention of doing all its work on it. If the company prospers and the workload grows, at a certain point the mainframe server will no longer be adequate. The only solutions are either to replace the mainframe server with a larger one (if it

exists) or to add a second mainframe server. Both of these can wreak major havoc on the company's operations. In contrast, with a distributed system, it may be possible simply to add more processors to the system, thus allowing it to expand gradually as the need arises. These advantages are summarized:

<u>Item</u>	<u>Description</u>
Economics	Microprocessors offer a better price/performance ratio than mainframe servers
Speed	A distributed system may have more total computing power than one
Inherent distribution	Some applications involve spatially separated machines
Reliability	If one machine crashes, the system as a whole can still survive
Incremental growth	Computing power can be added in small increments

A driving force is also the existence of large numbers of personal computers and the need for people to work together and share information in a convenient way without being bothered by geography or the physical distribution of people, data, and machines.

### **1.1.1.2 Advantages of Distributed Systems over Independent PCs**

Given that microprocessors are a cost-effective way to do business, why not just give everyone his own PC and let people work independently? For one thing, many users need to share data. For example, airline reservation clerks need access to the master data base of flights and existing reservations. Giving each clerk his own private copy of the entire data base would not work, since nobody would know which seats the other clerks had already sold. Shared data are absolutely essential to this and many other applications, so the machines must be interconnected. Interconnecting the machines leads to a distributed system.

Sharing often involves more than just data. Expensive peripherals, such as color laser printers, phototypesetters, and massive archival storage devices, are also candidates.

A third reason to connect a group of isolated computers into a distributed system is to achieve enhanced person-to-person communication.

Finally, a distributed system is potentially more flexible than giving each user an isolated personal computer. Although one model is to give each person a personal computer and connect them all with a LAN, this is not the only possibility. Another one is to have a mixture of personal and shared computers, perhaps of different sizes, and let jobs run on the most appropriate one, rather than always on the owner's machine. In this way, the workload can be spread over the computers more effectively, and the loss of a few machines may be compensated for by letting people run their jobs elsewhere.

<u>Item</u>	<u>Description</u>
Data sharing	Allow many users access to a common data base
Device sharing	Allow many users to share expensive devices
Communication	Make human-to-human communication easier
Flexibility	Spread the workload over available machines in the most cost effective way

### **1.1.1.3 Advantages of Distributed Computing Environment over Standalone Application**

The following are some of those key advantages:

- Higher performance. Applications can execute in parallel and distribute the load across multiple servers.
- Collaboration. Multiple applications can be connected through standard distributed computing mechanisms.
- Higher reliability and availability. Applications or servers can be clustered in multiple machines.
- Scalability. This can be achieved by deploying these reusable distributed components on powerful servers.
- Extensibility. This can be achieved through dynamic (re)configuration of applications that are distributed across the network.
- Higher productivity and lower development cycle time. By breaking up large problems into smaller ones, these individual components can be developed by smaller development teams in isolation.
- Reuse. The distributed components may perform various services that can potentially be used by multiple client applications. It saves repetitive development effort and improves interoperability between components.
- Reduced cost. Because this model provides a lot of reuse of once developed components that are accessible over the network, significant cost reductions can be achieved.

#### 1.1.1.4 Disadvantages of Distributed Systems

Although distributed systems have their strengths, they also have their weaknesses. We have already hinted at the worst problem: software. What kinds of operating systems, programming languages, and applications are appropriate for these systems? How much should the users know about the distribution? How much should the system do and how much should the users do?

A second potential problem is due to the communication network. It can lose messages, which requires special software to be able to recover, and it can become overloaded. When the network saturates, it must either be replaced or a second one must be added. Once the system comes to depend on the network, its loss or saturation can negate most of the advantages the distributed system was built to achieve.

The easy sharing of data, which we described above as an advantage, may turn out to be a two-edged sword. If people can conveniently access data all over the system, they may equally be able to conveniently access data that they have no business looking at. In other words, security is often a problem.

<i>Item</i>	<i>Description</i>
Software	Existing software for distributed systems is complex
Networking	The network can saturate or cause other problems
Security	Easy access also applies to secret data

The more components in a system, the greater the risk that the rest of the system will suffer unless special measures are taken in the event that one of the components fails. Special mechanisms are needed to avert these failures and make them transparent to the user. Moreover, the many components that make up a distributed system are potential sources of failures. Due to the physical and time separation, consistency (for example, with distributed databases) is more of a problem than with centralized systems. Leslie Lamport presents a (cynical) alternative characterization that highlights the complexity of distributed systems: A distributed system is one in which the failure of a computer you didn't even know existed can render your own computer unusable.

Another problem with these systems lies in the heterogeneity of their components. The larger the geographical area over which a distributed system is used, the greater the probability that different types of technology will be incorporated. Special mechanisms are needed to link together the different technologies in a network. The sprawling nature of a distributed system also raises a question about security, because the parts of a system that cannot be controlled are vulnerable to hackers and the like. Consequently, special mechanisms have to be introduced to safeguard security.

The following table summarizes the key advantages and disadvantages of distributed systems compared to centralized systems. Each application has to be looked at separately to determine whether there is a benefit to running it as a distributed application. Nevertheless, new concepts and tools are always required for working with distributed systems.

<i>Criteria</i>	<i>Centralized system</i>	<i>Distributed system</i>
Economics	low	high
Availability	low	high
Complexity	low	high
Consistency	simple	difficult
Scalability	poor	good
Technology	homogenous	heterogenous
Security	high	low

#### 1.1.2 Software Concepts

Although the hardware is important, the software is even more important. The image that a system presents to its users, and how they think about the system, is largely determined by the operating system software, not the hardware.

Operating systems cannot be put into nice, neat pigeonholes like hardware. By nature software is vague and amorphous. Still, it is more-or-less possible to distinguish two kinds of operating systems for multiple CPU systems: loosely coupled and tightly coupled. As we shall see, loosely and tightly-coupled software is roughly analogous to loosely and tightly-coupled hardware.

*Loosely-coupled software* allows machines and users of a distributed system to be fundamentally independent of one another, but still to interact to a limited degree where that is necessary. Consider a group of personal computers, each of which has its own CPU, its own memory, its own hard disk, and its own operating system, but which share some resources, such as laser printers and data bases, over a LAN. This system is loosely coupled, since the individual machines are clearly distinguishable, each with its own job to do. If the network should go down for some reason, the individual machines can still continue to run to a considerable degree, although some functionality may be lost.

Network operating systems are loosely-coupled software on loosely-coupled hardware. Other than the shared file system, it is quite apparent to the users that such a system consists of numerous computers. Each can run its own operating system and do whatever its owner wants. There is essentially no coordination at all, except for the rule that client-server traffic must obey the system's protocols.

The next evolutionary step beyond this is *tightly-coupled software* on the same loosely-coupled computers as a single timesharing system, rather than a collection of distinct machines. Some authors refer to this property as the single-system image. The essential idea is that the users should not have to be aware of the existence of multiple CPUs in the system. No current system fulfills this requirement entirely, but a number of candidates are on the horizon.

What are some characteristics of a distributed system? To start with, there must be a single, global interprocess communication mechanism so that any process can talk to any other process. It will not do to have different mechanisms on different machines or different mechanisms for local communication and remote must also be a global protection scheme.

Process management must also be the same everywhere. How processes are created, destroyed, started, and stopped must not vary from machine to machine. In short, the idea behind network operating systems, namely that any machine can do whatever it wants to as long as it obeys the standard protocols when engaging in client-server communication, is not enough. Not only must there be a single set of system calls available on all machines, but these calls must be designed so that they make sense in a distributed environment.

The file system must look the same everywhere, too. Having file names restricted to X characters in some locations and being unrestricted in others is undesirable. Also, every file should be visible at every location, subject to protection and security constraints, of course.

As a logical consequence of having the same system call interface everywhere, it is normal that identical kernels run on all the CPUs in the system. Doing so makes it easier to coordinate activities that must be global. For example, when a process has to be started up, all the kernels have to cooperate in finding the best place to execute it. In addition, a global file system is needed.

Nevertheless, each kernel can have considerable control over its own local resources. For example, since there is no shared memory, it is logical to allow each kernel to manage its own memory. For example, if swapping or paging is used, the kernel on each CPU is the logical place to determine what to swap or page. There is no reason to centralize this authority. Similarly, if multiple processes are running on some CPU, it makes sense to do the scheduling right there, too.

### **1.1.3 Design Issues**

In the preceding sections we have looked at distributed systems and related topics from both the hardware and software points of view. In the remainder of this chapter we will briefly look at some of the key design issues that people contemplating building a distributed system must deal with.

#### **1.1.3.1 Transparency**

The above characterization of distributed systems introduces them as a set of mutually cooperating processes. This aspect places the emphasis on the separation and, consequently, the possible distribution of the components of a system. This can prove to be a disadvantage to the applications programmer because of the additional complexity involved in dealing with distributed systems compared to centralized systems. It is desirable to conceal the complexity of distributed systems. In the literature this characteristic is described as *transparency*. Thus the complexity resulting from the distribution should be made transparent (i.e., invisible) to the applications

programmer. The aim behind this is to present a distributed system as a centralized system in order to simplify the distribution aspect.

Probably the single most important issue is how to achieve the single-system image. The collection of machines is seen as an old-fashioned timesharing system – a system that realizes this goal is often said to be transparent.

Transparency can be achieved at two different levels. Easiest to do is to hide the distribution from the users. For example, when a Unix user types make to recompile a large number of files in a directory, he need not be told that all the compilations are proceeding in parallel on different machines and are using a variety of file servers to do it. To him, the only thing that is unusual is that the performance of the system is halfway decent for a change. In terms of commands issued from the terminal and results displayed on the terminal, the distributed system can be made to look just like a single-processor system.

At a lower level, it is also possible, but harder, to make the system look transparent to programs. In other words, the system call interface can be designed so that the existence of multiple processors is not visible. Pulling the wool over the programmer's eyes is harder than pulling the wool over the terminal user's eyes, however.

What does transparency really mean? As an example, imagine a distributed system consisting of workstations each running some standard operating system. Normally, system services (e.g. reading files) are obtained by issuing a system call that traps to the kernel. In such a system, remote files should be accessed the same way. A system in which remote files are accessed by explicitly setting up a network connection to a remote server and then sending messages to it is not transparent because remote services are then being accessed differently than local ones. The programmer can tell that multiple machines are involved, and this is not allowed.

The concept of transparency can be applied to several aspects of a distributed system:

<i>Kind</i>	<i>Meaning</i>
Location transparency	The users cannot tell where resources are located
Migration transparency	Resources can move at will without changing their names
Replication transparency	The user cannot tell how many copies exist
Concurrency transparency	Multiple users can share resources automatically
Parallelism transparency	Activities can happen in parallel without users knowing
Access transparency	Identical ways in which access takes place to local/remote components
Failure transparency	Users are unaware of a failure of a component.
Technology transparency	Different technologies, such as programming languages and operating systems, are hidden from the user.

*Location transparency* refers to the fact that in a true distributed system, users cannot tell where hardware and software resources such as printers, files, and data bases are located. The name of the resource must not secretly encode the location of the resource, so names like machine1:prog.c or /machine1/prog.c are not acceptable.

*Migration transparency* means that resources must be free to move from one location to another without having their names change. Since a path like /work/news does not reveal the location of the server, it is location transparent. However, now suppose that the folks running the servers decide that reading network news really falls in the category "games" rather than in the category "work." Accordingly, they move news from server 2 to server 1. The next time client 1 boots and mounts the servers in his customary way, he will notice that /work/news no longer exists. Instead, there is a new entry. Thus the mere fact that a file or directory has migrated from one server to another has forced it to acquire a new name because the system of remote mounts is not migration transparent.

If a distributed system has *replication transparency*, the operating system is free to make additional copies of files and other resources on its own without the users noticing. Clearly, in the previous example, automatic replication is impossible because the names and locations are so closely tied together. To see how replication transparency might be achievable, consider a collection of  $n$  servers logically connected to form a ring. Each server maintains the entire directory tree structure but holds only a subset of the files themselves. To read a file, a client sends a message containing the full path name to any of the servers. That server checks to see if it has the file. If so, it returns the data requested. If not, it forwards the request to the next server in the ring, which then repeats the algorithm. In this system, the servers can decide by themselves to replicate any file on any or all servers, without the users having to know about it. Such a scheme is replication transparent because it allows the system to make copies of heavily used files without the users even being aware that this is happening.

Distributed systems usually have multiple, independent users. What should the system do when two or more users try to access the same resource at the same time? For example, what happens if two users try to update the same file at the same time? If the system is *concurrency transparent*, the users will not notice the existence of other users. One mechanism for achieving this form of transparency would be for the system to lock a resource automatically once someone had started to use it, unlocking it only when the access was finished. In this manner, all resources would only be accessed sequentially, never concurrently.

Finally, we come to the hardest one, *parallelism transparency*. In principle, a distributed system is supposed to appear to the users as a traditional, uniprocessor timesharing system. What happens if a programmer knows that his distributed system has 1000 CPUs and he wants to use a substantial fraction of them for a chess program that evaluates boards in parallel? The theoretical answer is that together the compiler, runtime system, and operating system should be able to figure out how to take advantage of this potential parallelism without the programmer even knowing it. Unfortunately, the current state-of-the-art is nowhere near allowing this to happen. Programmers who actually want to use multiple CPUs for a single problem will have to program this explicitly, at least for the foreseeable future. Parallelism transparency can be regarded as the holy grail for distributed systems designers. When that has been achieved, the work will have been completed, and it will be time to move on to new fields.

All this notwithstanding, there are times when users do not want complete transparency. For example, when a user asks to print a document, he often prefers to have the output appear on the local printer, not one 1000 km away, even if the distant printer is fast, inexpensive, can handle color and smell, and is currently idle.

It is costly and complicated to implement the mechanisms needed to meet the transparency criteria. For example, distributed systems have new types of fault situations that have to be dealt with that do not occur in centralized systems. As the degree of desired failure transparency increases, the mechanisms that implement the transparency become more complex. One of the main objectives of a middleware platform is to offer mechanisms that help to implement the above transparency criteria.

### 1.1.3.2 Flexibility

The second key design issue is flexibility. There are two schools of thought concerning the structure of distributed systems. One school maintains that each machine should run a traditional kernel that provides most services itself. The other maintains that the kernel should provide as little as possible, with the bulk of the operating system services available from level servers. These two models, known as the monolithic kernel and micro-kernel, respectively.

The monolithic kernel is basically today's centralized operating system augmented with networking facilities and the integration of remote services. Most system calls are made by trapping to the kernel, having the work there, and having the kernel return the desired result to the user process. With this approach, most machines have disks and manage their own local file systems. Many distributed systems that are extensions or imitations of Unix use this approach because Unix itself has a large, monolithic kernel.

If the monolithic kernel is the reigning champion, the microkernel is the up-and-coming challenger. Most distributed systems that have been designed from scratch use this method. The microkernel is more flexible because it does almost nothing.

### 1.1.3.3 Reliability

One of the original goals of building distributed systems was to make them more reliable than single-processor systems. The idea is that if a machine goes down, some other machine takes over the job. In other words, theoretically the overall system reliability could be the Boolean OR of the component reliabilities. For example, with four file servers, each with a 0.95 chance of being up at any instant, the probability of all four being down simultaneously is  $0.05^4 = 0.000006$ , so the probability of at least one being available is 0.999994, far better than that of any individual server.

That is the theory. The practice is that to function at all, current distributed systems count on a number of specific servers being up. As a result, some of them have an availability more closely related to the Boolean AND of the components than to the Boolean OR. In a widely-quoted remark, Lamport once defined a distributed system as "one on which I cannot get any work done because some machine I have never heard of has crashed." While this remark was (presumably) made somewhat tongue-in-cheek, there is clearly room for improvement here.

It is important to distinguish various aspects of reliability. *Availability*, as we have just seen, refers to the fraction of time that the system is usable. Lamport's system apparently did not score well in that regard. Availability can be enhanced by a design that does not require the simultaneous functioning of a substantial number of critical components. Another tool for improving availability is redundancy: key pieces of hardware and software should be replicated, so that if one of them fails the others will be able to take up the slack.

A highly reliable system must be highly available, but that is not enough. Data entrusted to the system must not be lost or garbled in any way, and if files are stored redundantly on multiple servers, all the copies must be kept consistent. In general, the more copies that are kept, the better the availability, but the greater the chance that they will be inconsistent, especially if updates are frequent. The designers of all distributed systems must keep this dilemma in mind all the time.

Another aspect of overall reliability is *security*. Files and other resources must be protected from unauthorized usage. Although the same issue occurs in single-processor systems, in distributed systems it is more severe. In a processor system, the user logs in and is authenticated. From then on, the system knows who the user is and can check whether each attempted access is legal. In a distributed system, when a message comes in to a server asking for something, the server has no simple way of determining who it is from. No name or identification field in the message can be trusted, since the sender may be lying. At the very least, considerable care is required here.

Still another issue relating to reliability is *fault tolerance*. Suppose that a server crashes and then quickly reboots. What happens? Does the server crash bring users down with it? If the server has tables containing important information about ongoing activities, recovery will be difficult at best.

In general, distributed systems can be designed to mask failures, that is, to hide them from the users. If a file service or other is actually constructed from a group of closely cooperating servers, it should be possible to construct it in such a way that users do not notice the loss of one or two servers, other than some performance degradation. Of course, the trick is to arrange this cooperation so that it does not add substantial overhead to the system in the normal case, when everything is functioning correctly.

#### 1.1.3.4 Performance

Always lurking in the background is the issue of performance. When running a particular application on a distributed system, it should not be appreciably worse than running the same application on a single processor. Unfortunately, achieving this is easier said than done.

Various performance metrics can be used. *Response time* is one, but so are *throughput* (number of jobs per hour), system utilization, and amount of network capacity consumed. Furthermore, the results of any benchmark are often highly dependent on the nature of the benchmark.

The performance problem is compounded by the fact that communication, which is essential in a distributed system (and absent in a single-processor system) is typically quite slow. Most of this time is due to unavoidable protocol handling on both ends, rather than the time the bits spend on the wire. Thus to optimize performance, one often has to minimize the number of messages. The difficulty with this strategy is that the best way to gain performance is to have many activities running in parallel on different processors, but doing so requires sending many messages.

One possible way out is to pay considerable attention to the grain size of all computations. Starting up a small computation remotely, such as adding two integers, is rarely worth it. On the other hand, starting up a long compute-bound job remotely may be worth the trouble. In general, jobs that involve a large number of small computations, especially ones that interact highly with one another, may cause trouble on a distributed system with relatively slow communication. Such jobs are said to exhibit *fine-grained parallelism*. On the other hand, jobs that involve large computations, low interaction rates, and little data, that is, coarse-grained parallelism, may be a better fit.

Fault tolerance also exacts its price. Good reliability is often best achieved by having several servers closely cooperating on a single request. For example, when a request comes in to a server, it could immediately send a copy of the message to one of its colleagues so that if it crashes before finishing, the colleague can take over. Naturally, when it is done, it must inform the colleague that the work has been completed, which takes another message. Thus we have at least two extra messages, which in the normal case cost time and network capacity and produce no tangible gain.

### **1.1.3.5 Scalability**

Most current distributed systems are designed to work with a few hundred CPUs. It is possible that future systems will be orders of magnitude larger, and solutions that work well for 200 machines will fail miserably for 200,000,000.

Next comes interactive access to all kinds of data bases and services, from electronic banking to reserving places in planes, trains, hotels, theaters, and restaurants, to name just a few. Before long, we have a distributed system with tens of millions of users. The question is: Will the methods currently developed scale to such large systems?

One guiding principle is clear: avoid centralized components, tables, and algorithms. Having a single mail server for 50 million users would not be a good idea. Even if it had enough CPU and storage capacity, the network capacity into and out of it would surely be a problem. Furthermore, the system would not tolerate faults well. A single power outage could bring the entire system down. Finally, most mail is local. Having a message sent by a user in Marseille to another user two blocks away pass through a machine in Paris is not the way to go.

Centralized tables are almost as bad as centralized components. How should one keep track of the telephone numbers and addresses of 50 million people? Suppose that each data record could be fit into 50 characters. A single 2.5-gigabyte disk would provide enough storage. But here again, having a single data base would undoubtedly saturate all the communication lines into and out of it. It would also be vulnerable to failures (a single speck of dust could cause a head crash and bring down the entire directory service). Furthermore, here too, valuable network capacity would be wasted shipping queries far away for processing.

Finally, centralized algorithms are also a bad idea. In a large distributed system, an enormous number of messages have to be routed over many lines. From a theoretical point of view, the optimal way to do this is collect complete information about the load on all machines and lines, and then run a graph theory algorithm to compute all the optimal routes. This information can then be spread around the system to improve the routing.

The trouble is that collecting and transporting all the input and output information would again be a bad idea for the reasons discussed above. In fact, any algorithm that operates by collecting information from all sites, sends it to a single machine for processing, and then distributes the results must be avoided.

Only decentralized algorithms should be used. These algorithms generally have the following characteristics, which distinguish them from centralized algorithms:

1. No machine has complete information about the system state.
2. Machines make decisions based only on local information.
3. Failure of one machine does not ruin the algorithm.
4. There is no implicit assumption that a global clock exists.

The first three follow from what we have said so far. The last is perhaps less obvious, but also important. Any algorithm that starts out with: "At precisely 12:00:00 all machines shall note the size of their output queue" will fail because it is impossible to get all the clocks exactly synchronized. Algorithms should take into account the lack of exact clock synchronization. The larger the system, the larger the uncertainty. On a single LAN, with considerable effort it may be possible to get all clocks synchronized down to a few milliseconds, but doing this nationally is tricky.

### **1.1.4 Middleware**

The preceding sections showed that distributed systems create new problems that do not exist in centralized systems. The question is how suitable concepts and mechanisms can be used to develop and execute applications in distributed systems. It is obvious that new concepts and mechanisms are necessary, but not at which level they should be embedded. In principle, different options exist—from support at the hardware level all the way to the extension of programming languages to enable support of distributed applications. Software solutions typically provide the greatest flexibility because of their suitability for integrating existing technologies (such as operating systems and programming languages).

These conditions lead to the concept of *middleware* that offers general services that support distributed execution of applications. The term *middleware* suggests that it is software positioned between the operating system and the application. Viewed abstractly, middleware can be envisaged as a “tablecloth” that spreads itself over a heterogeneous network, concealing the complexity of the underlying technology from the application being run on it.

#### 1.1.4.1 Middleware Tasks

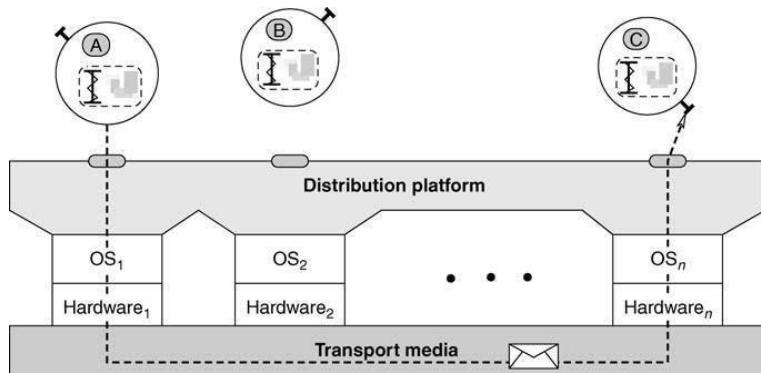
We start this section by looking at the tasks carried out by middleware. We will limit the discussion to the middleware support of an object-based application because the concepts of an object model ideally reflect the characteristics of distributed systems. An object encapsulates state and behavior and can only be accessed via a well-defined interface. The interface hides the details that are specific to the implementation, thereby helping to encapsulate different technologies. An object therefore becomes a unit of distribution. Recall that objects communicate with each other by exchanging messages.

We define the following middleware tasks:

- Object model support: Middleware should offer mechanisms to support the concepts incorporated in the object model.
- Operational interaction: Middleware should allow the operational interaction between two objects. The model used is the method invocation of an object-oriented programming language.
- Remote interaction: Middleware should allow the interaction between two objects located in different address spaces.
- Distribution transparency: From the standpoint of the program, interaction between objects is identical for both local and remote interactions.
- Technological independence: The middleware supports the integration of different technologies.

#### 1.1.4.2 The Structure of a Middleware Platform

As we mentioned earlier, middleware is conceptually located between the application and the operating system (see Figure 1.2). Because an object model serves as the underlying paradigm, the application is represented as a set of interacting objects. Each object is explicitly allocated to a hardware platform (i.e., we do not consider cases in which an object logically extends beyond computer boundaries).



*Fig. 1.2 Middleware for the support of object-based applications.*

The middleware hides the heterogeneity that occurs in a distributed system. This heterogeneity exists at different places:

- Programming languages: Different objects can be developed in different programming languages.
- Operating system: Operating systems have different characteristics and capabilities.
- Computer architectures: Computers differ in their technical details (e.g., data representations).
- Networks: Different computers are linked together through different network technologies.

Middleware overcomes this heterogeneity by offering equal functionality at all access points. Applications have access to this functionality through an Application Programming Interface (API). Because the API depends on the programming language in which the application is written, the API has to be adapted to the conditions of each programming language that is supported by the middleware.

An applications programmer typically sees middleware as a program library and a set of tools. The form these take naturally depends on the development environment that the programmer is using. Along with the programming language selected, this is also affected by the actual compiler/interpreter used to develop a distributed application.

### 1.1.4.3 Standardization of a Middleware

If we were to project a middleware to a global, worldwide network, we would find special characteristics that differ from those of a geographically restricted distributed system. At the global level, middleware spans several technological and political domains, and it can no longer be assumed that a homogenous technology exists within a distributed system.

Due to the heterogeneity and the complexity associated with it, we cannot assume that one vendor alone is able to supply middleware in the form of products for all environments. From the standpoint of market policy, it is generally desirable to avoid having the monopoly on a product and to support innovation through competition. However, the implementation of middleware through several competing products should not result in partial solutions that are not compatible.

Compatibility is only possible if all vendors of middleware adhere to a standard. The standard must therefore stipulate the *specification* for a product—an abstract description of a desired behavior that allows a degree of freedom in the execution of an implementation. It serves as a blueprint according to which different products (i.e., implementations) can be produced.

A specification as a standard identifies the verifiable characteristics of a system. If a system conforms to a standard, it is fulfilling these characteristics. From the view of the customer, standards offer a multitude of advantages. In an ideal situation a standard guarantees vendor independence, thereby enabling a customer to select from a range of products without having to commit to one particular vendor. For customers this means protection of the investments they have to make in order to use a product. The following characteristics are associated with *open standards*:

Nonproprietary: The standard itself is not subject to any commercial interests.

Freely available: Access to the standard is available to everyone.

Technology independent: The standard represents an abstraction of concrete technical mechanisms and only defines a system to the extent that is necessary for compatibility between products.

Democratic creation process: The creation and subsequent evolution of the standard is not ruled by the dominance of one company but takes place through democratic processes.

Product availability: A standard is only effective if products exist for it. In this respect a close relationship exists between a standard and the products that can technically be used in conjunction with the standard.

### 1.1.4.4 Portability and Interoperability

In the context of middleware, a standard has to establish the interfaces between different components to enable their interaction with one another. We want to distinguish between two types of interface: horizontal and vertical (see Figure 1.3). The horizontal interface exists between an application and the middleware and defines how an application can access the functionality of the middleware. This is also referred to as an *Application Programming Interface* (API). The standardization of the interface between middleware and application results in the *portability* of an application to different middleware because the same API exists at each access point.

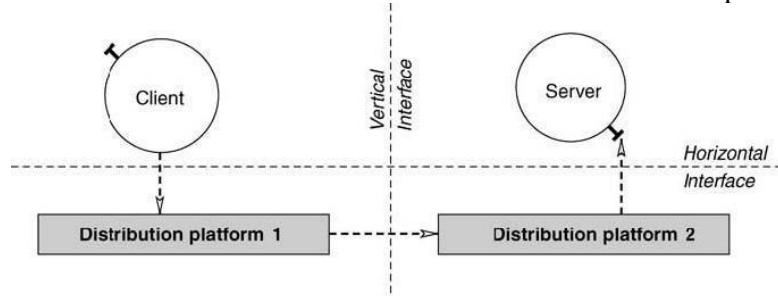


Fig. 1.3. Portability and interoperability.

In addition to the horizontal interface, there is a vertical interface that defines the interface between two instances of a middleware platform. This vertical interface is typically defined through a protocol on the basis of messages, referred to as *protocol data units* (PDUs). A PDU is a message sent over the network. Both client and server exchange PDUs to implement the protocol. The vertical interface separates technological domains and ensures that applications can extend beyond the influence area of the product of middleware. The standardization of this interface allows *interoperability* between applications.

Applications programmers are typically only interested in the horizontal interface because it defines the point of contact to their applications. From the view of the applications programmer, the vertical interface is of minor importance for the development of an application. Yet an implicit dependency exists between vertical and horizontal interfaces. For example, coding rules for the PDUs have to exist in the vertical interface for all data types available in the horizontal interface of an application.

### 1.1.5 Sample Application

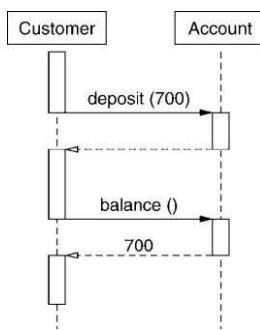
In this section we will present a simple application. The degree of completeness of the example is of minor importance for our discussion. It is intentionally kept simple in order to focus on middleware issues.

#### The Account Example

The first steps in any software development process are the analysis and design of the problem domain. The result of this process is a formal description of the application that is to be developed. One possible way to describe the application is through the *Unified Modeling Language* (UML). In the following we introduce a simple application where the design is trivial. Our emphasis is the distribution of the application with the help of a middleware, not the analysis and design process itself.

The application models a customer who wishes to do operations on a bank account. Here we are not concerned with different types of accounts, or how accounts are created by a bank. In order to keep the scenario simple, we assume that only one customer and one account exist, each represented through an object. The account maintains a balance, and the customer can deposit and withdraw money through appropriate operations. The customer can furthermore inquire as to the balance of the account.

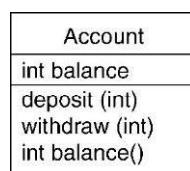
We use UML to further formalize the account application. UML introduces the notion of a *sequence diagram* that describes a use case of our application. For example, one possible use case might be that the customer deposits 700 units and then asks the account for the current balance. Assuming that the previous balance was 0, the new balance should be 700. This is only one of many possible use cases. For complex applications we would have several such use cases to describe the behavior of the application.



*Fig. 1.4 Sequence diagram for account use case.*

The use case described in the previous paragraph is depicted in Figure 1.4. The notation is based on the UML notation for sequence diagrams. The two actors—the customer and the account—are listed horizontally. Time flows from top to bottom. The arrows show invocations and responses between the two actors. The white bars indicate activity. In combination, the white bars and the arrows show clearly how the thread of execution is passed between the customer and the account.

The next step in designing the application is to decompose the problem domain into classes. For our simple account application, we will only introduce one class: Account. Figure 1.5 shows the class diagram for the account application. For the account class we only need one variable representing the balance of the account. The bottom section of the class diagram lists all the operations that instances of this class accept.



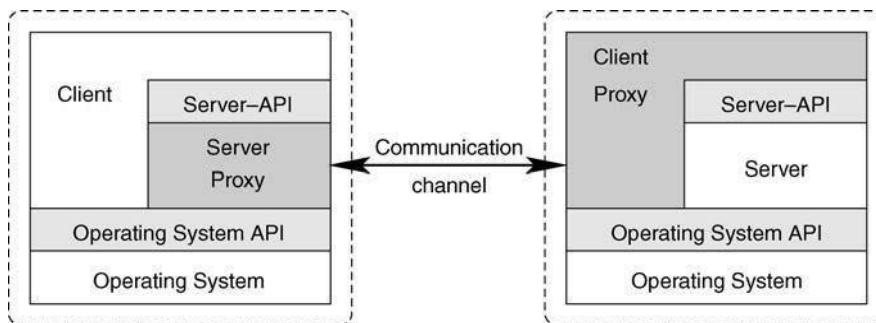
*Fig. 1.5 UML class diagram for the account example.*

## Distribution of the Sample Application

The server layer contains the account object. The client layer accesses this object through references (i.e., C++ pointers). It is important to note that the layers express a dependency relationship. Higher layers depend on lower layers. In that sense, the client depends on the server, but not vice versa.

The separation of client and server into different address spaces assumes that the actual parameters are being transmitted between processes since a common address space no longer exists. All data belonging to the parameters of an interaction between a client and a server must therefore be transmitted explicitly to the address space of the server. The data must be self-contained; that is, it is not allowed to contain a pointer that is only valid in the context of the client.

Figure 1.6 illustrates the principle with the client and server in different address spaces. A proxy of the server (component in dark gray shading) exists on the client side. This proxy offers the same API as the server itself. Its tasks include transmitting all current parameters over a communications channel to the remote address space.



*Fig. 1.6. Distributed execution of the application*

In the remote address space, a proxy of the client accepts the data and executes the actual invocation on the server. Outwardly it is not possible to distinguish the proxies from their “originals,” so the distribution of client and server is transparent. The proxies are used to fill the gaps on either side so that client and server are unaware of the separation.

### 1.1.6 Problems

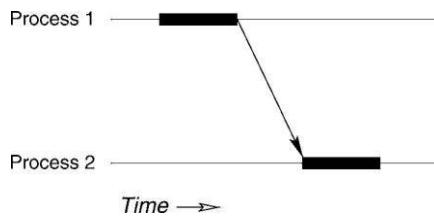
1. Name two advantages and two disadvantages of distributed systems over centralized ones.
2. The terms loosely-coupled system and tightly-coupled system are often used to described distributed computer systems. What is the different between them?
3. What is meant by a single-system image?
4. Suppose that you have a large source program consisting of  $m$  files to compile. The compilation is to take place on a system with processors, where  $n \gg m$ . The best you can hope for is an  $m$ -fold speedup over a single processor. What factors might cause the speedup to be less than this maximum?

## 1.2 Communication in Distributed Systems

The single most important difference between a distributed system and a uniprocessor system is the interprocess communication. In a uniprocessor system, most interprocess communication implicitly assumes the existence of shared memory. A typical example is the producer-consumer problem, in which one process writes into a shared buffer and another process reads from it. Even that most basic form of synchronization, the semaphore, requires that one word (the semaphore variable itself) is shared. In a distributed system there is no shared memory whatsoever, so the entire nature of interprocess communication must be completely rethought from scratch.

Processes are active components with a state and a behavior. The state consists of the data that is managed by the process. The behavior corresponds to the implementation of the applications logic. The processes cooperate with one another through message exchange. A message consists of a sequence of bytes that are transported between two processes via a communications medium. Certain rules apply to the way in which a message is structured internally so that it can be interpreted by the process received.

We limit our discussion to the communication that takes place between two processes. In message exchange one process assumes the role of sender and another process that of receiver. A space-time diagram can be used to visualize the process during a communications procedure. In Figure 1.7 time runs from left to right. Process 1 is the sender, and process 2 is the receiver.



**Fig. 1.7 Message exchange between two processes.**

A process is either in an active or a passive state (the active state is indicated in Figure 1.7 by the thick black lines). A process can only carry out calculations during an active state. Various events, such as the receipt of a message, can change the state of a process. In Figure 1.7, the sender is making some calculations and changes to the passive state when it sends a message. If the message arrives at the receiver, the receiver changes to an active phase and makes calculations based on the content of the message.

A simple classification scheme of communication mechanisms is the following. We first look at different patterns in the message flow between sender and receiver. The message exchange shown in Figure 1.7 is an example of *message-oriented communication*: The sender transmits a message to the receiver but does not wait for a reply. Message exchange takes place in only one direction. In the case of *request-oriented communication*, the receiver responds to the sender with a reply message. The communications process is not complete until the sender has received a reply to his request.

The synchronicity of a communication mechanism is orthogonal to the communications patterns from a message-oriented and request-oriented standpoint. The synchronicity describes the time separation between sender and receiver. In *synchronous communication* the sender is passive during the communications process until the message has arrived at the receiver. Conversely, during *asynchronous communication* the sender remains active after a message has been sent. With this kind of communication, a sender can transmit messages more quickly than the receiver is able to accept them, and thus the transport system must be capable of buffering messages. Although the buffering of messages in the transport system is not necessary in the case of synchronous communication, sender and receiver are still coupled to one another timewise. Asynchronous communication is better, however, at supporting the parallelism of processes.

The two communications patterns and two types of synchronization produce four categories for the classification of communication mechanisms (see Table 1.1).

**Table 1.1. Classification of communication mechanisms**

<i>Communications pattern</i>	<i>Level of synchronization</i>	
Message-oriented	Asynchronous no-wait-send	Synchronous rendezvous
Request-oriented	remote service invocation	remote procedure call

*Remote procedure call* (RPC) is an example of synchronous request-oriented communication. The sender sends a request to the receiver and is passive until the receiver delivers the results of the request. An example of asynchronous request-oriented communication is *remote service invocation* (RSI). During this type of communication, the sender remains active while the receiver is processing the request. Although RSI makes better use of the parallelism offered in distributed systems, RPC is based on popular programming paradigms and is therefore intuitive in its interpretation.

Datagram services are an example of asynchronous message-oriented communication. The sender transmits a message to the receiver without waiting for a reply or changing to the passive state. The rendezvous, however, is used in the synchronization of two processes and is an example of synchronous message-oriented communication. Synchronous communication is used in the rendezvous to establish a common (logical) time between sender and receiver.

## 1.2.1 The Client-Server Model

The idea behind this model is to structure the distributed system as a group of cooperating processes, called servers, that offer services to the users, called clients. The client sends a request message to the server asking for some service. The server does the work and the data requested or an error code indicating why the work could not be performed.

The primary advantage is the simplicity. The client sends a request and gets an answer. No connection has to be established before use. The reply message serves as the acknowledgement to the request.

The client/server model introduces two roles that can be assumed by processes: the role of *service user* (client) and the role of *service provider* (server). The distribution of roles implies an asymmetry in the distributed execution of an application. The server offers a service to which one or more clients have access. Here processes act as natural units in the distribution. In the context of distributed systems, the communication between client and server can be based on one of the mechanisms mentioned in the previous section. The client/server model only introduces roles that can be assumed by a process. At a given point in time, a process can assume the role of both client and server. This scenario occurs, for example, when the server is carrying out a task and delegates a subtask to a subordinate server.

The RPC offers a fundamental communication mechanism for client/server interaction. The client is the initiator of an RPC, and the server provides the implementation of the remotely executed procedure. The request message contains all current input parameters for the procedure call. Conversely, the response message contains all results for the corresponding request produced by the server. The advantage of using remote procedure call as a communication mechanism for the client/server model is that it incorporates procedural programming paradigms and is therefore easily understood. The implementation of the procedure is an integral part of the server, and the invocation of the procedure is part of the application running in the client.

An advantage of the client/server model is the intuitive splitting of applications into a client part and a server part. Based on conventional abstractions such as procedural programming, it simplifies the design and the development of distributed applications. Over and above this, it makes it easy to migrate or integrate existing applications into a distributed environment. The client/server model also makes effective use of resources when a large number of clients are accessing a high-performance server. Another advantage of the client/server model is the potential for concurrency.

From a different point of view, all these advantages could also be considered disadvantages. For example, the restriction to procedural programming paradigms excludes other approaches such as functional or declarative programming. Furthermore, even procedural paradigms cannot always ensure that transparency is maintained between local and remote procedure calls since transparency can no longer be achieved in the case of radical system failure. The concurrency mentioned earlier as an advantage can also lead to problems because of its requirement that processes be synchronized.

Due to this simple structure, the communication services provided by the middleware can, for example, be reduced to two system calls, one for sending messages and one for receiving them. These system calls can be invoked through library procedures, say, `send(dest, &mptr)` and `receive(addr, &mptr)`. The former sends the message pointed to by `mptr` to a process identified by `dest` and causes the caller to be blocked until the message has been sent. The latter causes the caller to be blocked until a message arrives. When one does, the message is copied to the buffer pointed to by `mptr` and the caller is unblocked. The `addr` parameter specifies the address to which the receiver is listening. Many variants of these two procedures and their parameters are possible.

### 1.2.1.1 Addressing

In order for a client to send a message to a server, it must know the server's address.

One common scheme is to use two part names, specifying both a machine and a process number. Thus 243.4 or 4@243 or something similar designates process 4 on machine 243. The machine number is used by the kernel to get the message correctly delivered to the proper machine, and the process number is used by the kernel on that machine to determine which process the message is intended for. A nice feature of this approach is that every machine can number its processes starting at 0.

Nevertheless, machine.process addressing is far from ideal. Specifically, it is not transparent since the user is obviously aware of where the server is located, and transparency is one of the main goals of building a distributed system. To see why this matters, suppose that the file server normally runs on machine 243, but one day that machine is down. Machine 176 is available, but programs previously compiled using all have the number 243 built into them, so they will not work if the server is unavailable. Clearly, this situation is undesirable.

An alternative approach is to assign each process a unique address that does not contain an embedded machine number. One way to achieve this goal is to have a centralized process address that simply maintains a counter. Upon receiving a request for an address, it simply returns the current value of the counter and then increments it by one. The disadvantage of this scheme is that centralized components like this do not scale to large systems and thus should be avoided.

Yet another method for assigning process identifiers is to let each process pick its own identifier from a large, sparse address space, such as the space of 64-bit binary integers. The probability of two processes picking the same number is tiny, and the system scales well. However, here, too, there is a problem: How does the sending kernel know what machine to send the message to? On a LAN that supports broadcasting, the sender can broadcast a special locate packet containing the address of the destination process. Because it is a broadcast packet, it will be received by all machines on the network. All the kernels check to see if the address is theirs, and if so, send back a here I am message giving their network address (machine number). The sending kernel then uses this address, and furthermore, caches it, to avoid broadcasting the next time the server is needed.

Although this scheme is transparent, even with caching, the broadcasting puts extra load on the system. This extra load can be avoided by providing an extra machine to map (i.e. ASCII) service names to machine addresses. When this system is employed, processes such as servers are referred to by ASCII strings, and it is these strings that are embedded in programs, not binary machine or process numbers. Every time a client runs, on the first attempt to use a server, the client sends a query message to a special mapping server, often called a *name server*, asking it for the machine number where the server is currently located. Once this address has been obtained, the request can be sent directly. As in the previous case, addresses can be cached.

In summary, we have the following methods for addressing processes:

1. Hardwire machine.number into client code.
2. Let processes pick random addresses; locate them by broadcasting.
3. Put ASCII server names in clients; look them up at run time.

Each of these has problems. The first one is not transparent, the second one generates extra load on the system, and the third one requires a centralized component, the name server. Of course, the name server can be replicated, but doing so introduces the problems associated with keeping them consistent.

A completely different approach is to use special hardware. Let processes pick random addresses. However, instead of locating them by broadcasting, the network interface chips have to be designed to allow processes to store process addresses in them. Frames would then use process addresses instead of machine addresses. As each frame came by, the network interface chip would simply examine the frame to see if the destination process was on its machine. If so, the frame would be accepted; otherwise, it would not be.

### 1.2.1.2 Blocking versus Nonblocking Primitives

The message-passing primitives we have described so far are what are called blocking primitives (sometimes called synchronous primitives). When a process calls send it specifies a destination and a buffer to send to that destination. While the message is being sent, the sending process is blocked suspended. The instruction following the call to send is not executed until the message has been completely sent. Similarly, a call to receive does not return control until a message has actually been received and put in the message buffer pointed to by the parameter. The process remains suspended in receive until a message arrives, even if it takes hours. In some systems, the receiver can specify from whom it wishes to receive, in which case it remains blocked until a message from that sender arrives.

An alternative to blocking primitives are nonblocking primitives (some-times called asynchronous primitives). If send is nonblocking, it returns control to the caller immediately, before the message is sent. The advantage of this scheme is that the sending process can continue computing in parallel with the message transmission, instead of having the CPU go idle (assuming no other process is runnable). The choice between blocking and nonblocking primitives is normally made by the system designers (i.e. either one primitive is available or the other), although in a few systems both are available and users can choose their favorite.

However, the performance advantage offered by nonblocking primitives is offset by a serious disadvantage: the sender cannot modify the message buffer until the message has been sent. The consequences of the process overwriting the message during transmission are too horrible to contemplate. Worse yet, the sending process has no idea of when the transmission is done, so it never knows when it is safe to reuse the buffer. It can hardly avoid touching it forever.

There are two possible ways out. The first solution is to have the kernel copy the message to an internal kernel buffer and then allow the process to continue. From the sender's point of view, this scheme is the same as a blocking call: as soon as it gets control back, it is free to reuse the buffer. Of course, the message will not yet have been sent, but the sender is not hindered by this fact. The disadvantage of this method is that every outgoing message has to be copied from user space to kernel space.

The second solution is to interrupt the sender when the message has been sent to inform it that the buffer is once again available. No copy is required here, which saves time, but user-level interrupts make programming tricky, difficult, and subject to race conditions, which makes them irreproducible. Most experts agree that although this method is highly efficient and allows the most parallelism, the disadvantages greatly outweigh the advantages: programs based on interrupts are difficult to write correctly and nearly impossible to debug when they are wrong.

Sometimes the interrupt can be disguised by starting up a new thread of control within the sender's address space. Although this is somewhat cleaner than a raw interrupt, it is still far more complicated than synchronous communication. If only a single thread of control is available, the choices come down to:

1. Blocking send (CPU idle during message transmission).
2. Nonblocking send with copy (CPU time wasted for the extra copy).
3. Nonblocking send with interrupt (makes programming difficult).

Under normal conditions, the first choice is the best. It does not maximize the parallelism, but is simple to understand and simple to implement. It also does not require any kernel buffers to manage. Furthermore, the message will usually be out the door faster if no copy is required. On the other hand, if overlapping processing and transmission are essential for some application, a nonblocking send with copying is the best choice.

The essential difference between a synchronous primitive and an asynchronous one is whether the sender can reuse the message buffer immediately after getting control back without fear of messing up the send. When the message actually gets to the receiver is irrelevant.

In the alternative view, a synchronous primitive is one in which the sender is blocked until the receiver has accepted the message and the acknowledgement has gotten back to the sender. Everything else is asynchronous in this view. There is complete agreement that if the sender gets control back before the message has been copied or sent, the primitive is asynchronous. Similarly, everyone agrees that when the sender is blocked until the receiver has acknowledged the message, we have a synchronous primitive.

The disagreement comes on whether the intermediate cases (message copied or copied and sent, but not acknowledged) counts as one or the other. Operating systems designers tend to prefer our way, since their concern is with buffer management and message transmission. Programming language designers tend to prefer the alternative definition, because that is what counts at the language level.

Just as send can be blocking or nonblocking, so can receive. A nonblocking receive just tells the kernel where the buffer is, and returns control almost immediately. Again here, how does the caller know when the operation has completed? One way is to provide an explicit wait primitive that allows the receiver to block when it wants to. Alternatively (or in addition to wait), the designers may provide a test primitive to allow the receiver to poll the kernel to check on the status. A variant on this idea is a conditional-receive, which either gets a message or signals failure, but in any event returns immediately, or within some interval. Finally, here too, interrupts can be used to signal completion. For the most part, a blocking version of receive is much simpler and greatly preferred.

If multiple threads of control are present within a single address space, the arrival of a message can cause a thread to be created spontaneously.

An issue closely related to blocking versus nonblocking calls is that of timeouts. In a system in which send calls block, if there is no reply, the sender will block forever. To prevent this situation, in some systems the caller may specify a time interval within which it expects a reply. If none arrives in that interval, the send call terminates with an error status.

### 1.2.1.3 Implementing the Client-Server Model

We have looked at four design issues, addressing, blocking, buffering, and reliability, each with several options. The major alternatives are summarized in:

<i>Item</i>	<i>Option 1</i>	<i>Option 2</i>	<i>Option 3</i>
Addressing	Machine number	Sparse process addresses	ASCII names looked up via server
Blocking	Blocking primitives	Nonblocking with copy to kernel	Nonblocking with interrupt
Buffering	Unbuffered, discarding unexpected messages	Unbuffered, temporarily keeping unexpected messages	Mailboxes
Reliability	Unreliable	Request-Ack-Reply Ack	Request-Reply-Ack

While the details of how message passing is implemented depend to some extent on which choices are made, it is still possible to make some general comments about the implementation, protocols, and software. To start with, virtually all networks have a maximum packet size, typically a few thousand bytes at most. Messages larger than this must be split up into multiple packets and sent separately. Some of these packets may be lost or garbled, and they may even arrive in the wrong order. To deal with this problem, it is usually sufficient to assign a message number to each message, and put it in each packet belonging to the message, along with a sequence number giving the order of the packets.

However, an issue that still must be resolved is the use of acknowledgements. One strategy is to acknowledge each individual packet. Another one is to acknowledge only entire messages. The former has the advantage that if a packet is lost, only that packet has to be retransmitted, but it has the disadvantage of requiring more packets on the network. The latter has the advantage of fewer packets, but the disadvantage of a more complicated recovery when a packet is lost (because a client requires retransmitting the entire message). The choice depends largely on the loss rate of the network being used.

## 1.2.2 Remote Procedure Call (RPC)

Although the client-server model provides a convenient way to structure a distributed operating system, it suffers from one incurable flaw: the basic paradigm around which all communication is built is send and receive are fundamentally engaged in doing of the key concepts of centralized systems, making it the basis for distributed computing has struck many workers in the field as a mistake. Their goal is to make distributed computing look like centralized computing. Building everything around I/O is not the way to do it.

Another approach is to allow programs to call procedures located on other machines. When a process on machine A calls a procedure on machine B, the calling process on A is suspended, and execution of the called procedure takes place on B. Information can be transported from the caller to the callee in the parameters and can come back in the procedure result. No message passing or I/O at all is visible to the programmer. This method is known as remote procedure call, or often just RPC.

While the basic idea sounds simple and elegant, subtle problems exist. To start with, because the calling and called procedures run on different machines, they execute in different address spaces, which causes complications. Parameters and results also have to be passed, which can be complicated, especially if the machines are not identical. Finally, both machines can crash, and each of the possible failures causes different problems. Still, most of these can be dealt with, and RPC is a widely-used technique that underlies many distributed operating systems.

### 1.2.2.1 Basic RPC Operation

To understand how RPC works, it is important first to fully understand how a conventional (i.e. single machine) procedure call works. Consider a call like

```
count = read(fd, buf, nbytes);
```

where *fd* is an integer, *buf* is an array of characters, and *nbytes* is another integer. To make the call, the caller pushes the parameters onto the stack in order, last one first (The reason that C compilers push the parameters in reverse order has to do with printf – by doing so, printf can always locate its first parameter, the format string.) After read has finished running, it puts the return value in a register, removes the return address, and transfers control back to the caller. The caller then removes the parameters from the stack, returning it to the original state.

Several things are worth noting. For one, in C, parameters can be call-by-value or call-by-reference. A value parameter, such as fd or nbytes, is simply copied to the stack. To the called procedure, a value parameter is just an initialized local variable. The called procedure may modify it, but such changes do not affect the original value at the calling side.

A reference parameter in C is a pointer to a variable (i.e. the address of the variable), rather than the value of the variable. In the call to read, the second parameter is a reference parameter because arrays are always passed by reference in C. What is actually pushed onto the stack is the address of the character array. If the called procedure uses this parameter to store something into the character array, it does modify the array in the calling procedure. The difference between call-by-value and call-by-reference is quite important for RPC, as we shall see.

One other parameter passing mechanism also exists, although it is not used in C. It is called call-by-copy/restore. It consists of having the variable copied to the stack by the caller, as in call-by-value, and then copied back after the call, overwriting the caller's original value. Under most conditions, this achieves the same effect as call-by-reference, but in some situations, such as the same parameter being present multiple times in the parameter list, the semantics are different.

The decision of which parameter passing mechanism to use is normally made by the language designers and is a fixed property of the language. Sometimes it depends on the data type being passed. In C, for example, integers and other scalar types are always passed by value, whereas arrays are always passed by reference.

The idea behind RPC is to make a remote procedure call look as much as possible like a local one. In other words, we want RPC to be transparent—the calling procedure should not be aware that the called procedure is executing on a different machine, or vice versa. Suppose that a program needs to read some data from a file. The programmer puts a call to read in the code to get the data. In a traditional (single-processor) system, the read routine is extracted from the library by the linker and inserted into the object program. It is a short procedure, usually written in assembly language, that puts the parameters in registers and then issues a READ system call by trapping to the kernel. In essence, the read procedure is a kind of interface between the user code and the operating system.

Even though read issues a kernel trap, it is called in the usual way, by pushing the parameters onto the stack. Thus the programmer does not know that read is actually doing something fishy.

RPC achieves its transparency in an analogous way. When *read* is actually a remote procedure (e.g. one that will run on the file server's machine), a different version of *read*, called a *client stub*, is put into the library. Like the original one, it too, is called using the calling sequence. Also like the original one, it too, traps to the kernel. Only unlike the original one, it does not put the parameters in registers and ask the kernel to give it data. Instead, it packs the parameters into a message and asks the kernel to send the message to the server. Following the call to send, the client stub calls receive, blocking itself until the reply comes back.

When the message arrives at the server, the kernel passes it up to a *server stub* that is bound with the actual server. Typically the server stub will have called receive and be blocked waiting for incoming messages. The server stub unpacks the parameters from the message and then calls the server procedure in the usual way. From the server's point of view, it is as though it is being called directly by the client—the parameters and return address are all on the stack where they belong and nothing seems unusual. The server performs its work and then returns the result to the caller in the usual way. For example, in the case of read, the server will fill the buffer, pointed to by the second parameter, with the data. This buffer will be internal to the server stub.

When the server stub gets control back after the call has completed, it packs the result (the buffer) in a message and calls send to return it to the client. Then it goes back to the top of its own loop to call receive, waiting for the next message.

When the message gets back to the client machine, the kernel sees that it is addressed to the client process (to the stub part of that process, but the kernel does not know that). The message is copied to the waiting buffer and the client process unblocked. The client stub inspects the message, unpacks the result, copies it to its caller, and returns in the usual way. When the caller gets control following the call to read, all it knows is that its data are available. It has no idea that the work was done remotely instead of by the local kernel.

This blissful ignorance on the part of the client is the beauty of the whole scheme. As far as it is concerned, remote services are accessed by making ordinary (i.e. local) procedure calls, not by calling send and receive.

All the details of the message passing are hidden away in the two library procedures, just as the details of actually making system call traps are hidden away in traditional libraries.

To summarize, a remote procedure call occurs in the following steps:

1. The client procedure calls the client stub in the normal way.
2. The client stub builds a message and traps to the kernel.
3. The kernel sends the message to the remote kernel.
4. The remote kernel gives the message to the server stub.
5. The server stub unpacks the parameters and calls the server.
6. The server does the work and returns the result to the stub.
7. The server stub packs it in a message and traps to the kernel.
8. The remote kernel sends the message to the client's kernel.
9. The client's kernel gives the message to the client stub.
10. The stub unpacks the result and returns to the client.

The net effect of all these steps is to convert the local call by the client procedure to the client stub to a local call to the server procedure without either client or server being aware of the intermediate steps.

### 1.2.2.2 Parameter Passing

The function of the client stub is to take its parameters, pack them into a message, and send it to the server stub. While this sounds straightforward, it is not quite as simple as it at first appears. In this section we will look at some of the issues concerned with parameter passing in RPC systems. Packing parameters into a message is called *parameter marshaling*.

As the simplest possible example, consider a remote procedure, *sum(i,j)*, that takes two integer parameters and returns their arithmetic sum. (As a practical matter, one would not normally make such a simple procedure remote due to the overhead, but as an example it will do.). The client stub takes its two parameters and puts them in a message as indicated. It also puts the name or number of the procedure to be called in the message because the server might support several different calls, and it has to be told which one is required.

When the message arrives at the server, the stub examines the message to see which procedure is needed, and then makes the appropriate call. If the server also supports the remote procedures difference, product, and quotient, the server stub might have a switch statement in it, to select the procedure to be called, depending on the first field of the message. The actual call from the stub to the server looks much like the original client call, except that the parameters are variables initialized from the incoming message, rather than constants.

When the server has finished, the server stub gains control again. It takes the result, provided by the server, and packs it into a message. This message is sent back to the client stub, which unpacks it and returns the value to the client procedure.

As long as the client and server machines are identical and all the parameters and results are scalar types, such as integers, characters, and Booleans, this model works fine. However, in a large distributed system, it is common that multiple machine types are present. Each machine often has its own representation for numbers, characters, and other data items.

Similar problems can occur with the representation of integers (1's complement versus 2's complement), and especially with floating-point numbers. In addition, an even more annoying problem exists because some machines (e.g. Intel), number their bytes from right to left (format named *little endian*), whereas others (e.g. Sun), number them the other way (format named *big endian*) – the format names are given after the politicians in Gulliver's Travels who went to war over which end of an egg to break. Once a standard has been agreed upon for representing each of the basic data types, given a parameter list and a message, it is possible to deduce which bytes belong to which parameter, and thus to solve the problem.

We might decide to transmit a character in the rightmost byte of a word (leaving the next 3 bytes empty), a float as a whole word, and an array as a group of words equal to the array length, preceded by a word giving the length. Thus given these rules, the client stub for foobar knows that it must use the format and the server stub knows that incoming messages for foobar will have the format. Having the type information for the parameters makes it possible to make any necessary conversions.

Even with this additional information, there are still some issues open. In particular, how should information be represented in the messages? One way is to devise a network standard or canonical form for integers, characters, Booleans, floating-point numbers, and so on, and require all senders to convert their internal representation to this form while marshaling.

The problem with this method is that it is sometimes inefficient. Suppose that a big endian client is talking to a big endian server. According to the rules, the client must convert everything to little endian in the message, and the server must convert it back again when it arrives. Although this is unambiguous, it requires two conversions when in fact none were necessary. This observation gives rise to a second approach: the client uses its own native format and indicates in the first byte of the message which format this is. Thus a little endian client builds a little endian message and a big endian client builds a big endian message. As soon as a message comes in, the server stub examines the first byte to see what the client is. If it is the same as the server, no conversion is needed. Otherwise, the server stub converts everything. Although we have only discussed converting from one endian to the other, conversions between one's and two's complement, and so on, can be handled in the same way. The trick is knowing what the message layout is and what the client is. Once these are known, the rest is easy (provided that everyone can convert from everyone else's format).

Now we come to the question of where the stub procedures come from. In many RPC-based systems, they are generated automatically. As we have seen, given a specification of the server procedure and the encoding rules, the message format is uniquely determined. Thus it is possible to have a compiler read the server specification and generate a client stub that packs its parameters into the officially approved message format. Similarly, the compiler can also produce a server stub that unpacks them and calls the server. Having both stub procedures generated from a single formal specification of the server not only makes life easier for the programmers, but reduces the chance of error and makes the system transparent with respect to differences in internal representation of data items.

Finally, we come to our last and most difficult problem: How are pointers passed? The answer is: only with the greatest of difficulty, if at all. Remember that a pointer is meaningful only within the address space of the process in which it is being used. Getting back to our read example discussed earlier, if the second parameter (the address of the buffer) happens to be 1000 on the client, one cannot just pass the number 1000 to the server and expect it to work. Address 1000 on the server might be in the middle of the program text.

One solution is just to forbid pointers and reference parameters in general. However, these are so important that this solution is highly undesirable. In fact, it is not necessary either. In the read example, the client stub knows that the second parameter points to an array of characters. Suppose, for the moment, that it also knows how big the array is. One strategy then becomes apparent: copy the array into the message and send it to the server. The server stub can then call the server with a pointer to this array, even though this pointer has a different numerical value than the second parameter of read has. Changes the server makes using the pointer (e.g. storing data into it) directly affect the message buffer inside the server stub. When the server finishes, the original message can be sent back to the client stub, which then copies it back to the client. In effect, call-by-reference has been replaced by copy/restore. Although this is not always identical, it frequently is good enough.

One optimization makes this mechanism twice as efficient. If the stubs know whether the buffer is an input parameter or an output parameter to the server, one of the copies can be eliminated. If the array is input to the server (e.g. in a call to write) it need not be copied back. If it is output, it need not be sent over in the first place. The way to tell them is in the formal specification of the server procedure. Thus associated with every remote procedure is a formal specification of the procedure, written in some kind of specification language, telling what the parameters are, which are input and which are output (or both), and what their (maximum) sizes are. It is from this formal specification that the stubs are generated by a special stub compiler.

As a final comment, it is worth noting that although we can now handle pointers to simple arrays and structures, we still cannot handle the most general case of a pointer to an arbitrary data structure such as a complex graph. Some systems attempt to deal with this case by actually passing the pointer to the server stub and generating special code in the server procedure for using pointers.

Normally, a pointer is followed (dereferenced) by putting it in a register and indirecting through the register. When this special technique is used, a pointer is dereferenced by sending a message back to the client stub asking it to fetch and send the item being pointed to (reads) or store a value at the address pointed to (writes). While this method works, it is often highly inefficient. Imagine having the file server store the bytes in the buffer by sending back each one in a separate message. Still, it is better than nothing, and some systems use it.

### 1.2.2.3 Dynamic Binding

An issue that we have glossed over so far is how the client locates the server. One method is just to the network address of the server into the client. The trouble with this approach is that it is extremely inflexible. If the server moves or if the server is replicated or if the interface changes, numerous programs will have to be found and recompiled. To avoid all these problems, some distributed systems use what is called *dynamic binding* to match up clients and servers. In this section we will describe the ideas behind dynamic binding.

The starting point for dynamic binding is the server's formal specification. As an example, consider the server with the specification that tells the name of the server (e.g. file\_server), the version number (e.g. 3.1), and a list of procedures provided by the server (read, write, create, and delete).

For each procedure, the types of the parameters are given. Each parameter is specified as being an in parameter, an out parameter, or an in out parameter. The direction is relative to the server. An in parameter, such as the file name, name, is sent from the client to the server. This one is used to tell the server which file to read from, write to, create, or delete. Similarly, bytes tells the server how many bytes to transfer and position tells where in the file to begin reading or writing. An out parameter such as buf in read, is sent from the server to the client. Buf is the place where the file server puts the data that the client has requested. An in out parameter, of which there are none in this example, would be sent from the client to the server, modified there, and then sent back to the client (copy/restore) is typically used for pointer parameters in cases where the server both reads and modifies the data structure being pointed to. The directions are crucial, so the client stub knows which parameters to send to the server, and the server stub knows which ones to send back.

As we pointed out earlier, this particular example is a stateless server. For a Unix-like server, one would have additional procedures open and close, and different parameters for read and write. The concept of RPC itself is neutral, permitting the system designers to build any kind of servers they desire.

The primary use of the formal specification is as input to the stub generator, which produces both the client stub and the server stub. Both are then put into the appropriate libraries. When a user (client) program calls any of the procedures defined by this specification, the corresponding client stub procedure is linked into its binary. Similarly, when the server is compiled, the server stubs are linked with it too.

When the server begins executing, the call to initialize outside the main loop *exports* the server interface. What this means is that the server sends a message to a program called a *binder*, to make its existence known. This process is referred to as *registering* the server. To register, the server gives the binder its name, its version number, a unique identifier, typically 32 bits long, and a *handle* used to locate it. The handle is system dependent, and might be an Ethernet address, an IP address, an X.500 address, a sparse process identifier, or something else. In addition, other information, for example, concerning authentication, might also be supplied. A server can also deregister with the binder when it is no longer prepared to offer service. The binder interface is:

<i>Call</i>	<i>Input</i>	<i>Output</i>
Register	Name,version,handle, unique id	
Deregister	Name,version,unique id	
Lookup	Name, version	Handle, unique id

Given this background, now consider how the client locates the server. When the client calls one of the remote procedures for the first time, say, read, the client stub sees that it is not yet bound to a server, so it sends a message to the binder asking to import version 3.1 of the file\_server interface. The binder checks to see if one or more servers have already exported an interface with this name and version number. If no currently running server is willing to support this interface, the read call fails. By including the version number in the matching process, the binder can ensure that clients using obsolete interfaces will fail to locate a server rather than locate one and get unpredictable results due to incorrect parameters.

On the other hand, if a suitable server exists, the binder gives its handle and unique identifier to the client stub. The client stub uses the handle as the address to send the request message to. The message contains the parameters and the unique identifier, which the server's kernel uses to direct the incoming message to the correct server in the event that several servers are running on that machine.

This method of exporting and importing interfaces is highly flexible. For example, it can handle multiple servers that support the same interface. The binder can spread the clients randomly over the servers to even the load if it

wants to. It can also poll the servers periodically, automatically deregistering any server that fails to respond, to achieve a degree of fault tolerance. Furthermore, it can also assist in authentication. A server could specify, for example, that it only wished to be used by a specific list of users, in which case the binder would refuse to tell users not on the list about it. The binder can also verify that both client and server are using the same version of the interface.

However, this form of dynamic binding also has its disadvantages. The extra overhead of exporting and importing interfaces costs time. Since many client processes are short lived and each process has to start all over again, the effect may be significant. Also, in a large distributed system, the binder may become a bottleneck, so multiple binders are needed. Consequently, whenever an interface is registered or deregistered, a substantial number of messages will be needed to keep all the binders synchronized and up to date, creating even more overhead.

#### 1.2.2.4 RPC Protocols

Theoretically, any old protocol will do as long as it gets the bits from the client's kernel to the server's kernel, but practically there are several major decisions to be made here, and the choices made can have a major impact on the performance. The first decision is between a *connection-oriented protocol* and a *connectionless protocol*. With a connection-oriented protocol, at the time the client is bound to the server, a connection is established between them. All traffic, in both directions, uses this connection.

The advantage of having a connection is that communication becomes much easier. When a kernel sends a message, it does not have to worry about it getting lost, nor does it have to deal with acknowledgements. All that is handled at a lower level, by the software that supports the connection. When operating over a wide-area network, this advantage is often too strong to resist.

The disadvantage, especially over a LAN, is the performance loss. All that extra software gets in the way. Besides, the main advantage (no lost packets) is hardly needed on a LAN, since LANs are so reliable. As a consequence, most distributed operating systems that are intended for use in a single building or campus use connectionless protocols.

The second major choice is whether to use a standard general-purpose protocol or one specifically designed for RPC. Since there are no standards in this area, using a custom RPC protocol often means designing your own (or borrowing a friend's). System designers are split about evenly on this one.

Some distributed systems use IP (or UDP, which is built on IP) as the basic protocol. This choice has several things going for it:

1. The protocol is already designed, saving considerable work.
2. Many implementations are available, again saving work.
3. These packets can be sent and received by nearly all systems.
4. IP and UDP packets are supported by many existing networks.

In short, IP and UDP are easy to use and fit in well with existing Unix systems and networks such as the Internet. This makes it straightforward to write clients and servers that run on Unix systems, which certainly aids in getting code running quickly and in testing it.

As usual, the downside is the performance. IP was not designed as an user protocol. It was designed as a base upon which reliable TCP connections could be established over recalcitrant internetworks. For example, it can deal with gateways that fragment packets into little pieces so they can pass through networks with a tiny maximum packet size. Although this feature is never needed in a LAN-based distributed system, the packet header fields dealing with fragmentation have to be filled in by the sender and verified by the receiver to make them legal IP packets. IP packets have in total 13 header fields, of which three are useful: the source and destination addresses and the packet length. The other 10 just come along for the ride, and one of them, the header checksum, is time consuming to compute. To make matters worse, UDP has another checksum, covering the data as well.

The alternative is to use a specialized RPC protocol that, unlike IP, does not attempt to deal with packets that have been bouncing around the network for a few minutes and then suddenly materialize out of thin air at an inconvenient moment. Of course, the protocol has to be invented, implemented, tested, and embedded in existing systems, so it is considerably more work. Furthermore, the rest of the world tends not to jump with joy at the birth of yet another new protocol. In the long run, the development and widespread acceptance of a high-performance RPC protocol is definitely the way to go, but we are not there yet.

### 1.2.2.5 Failure Semantics

Ideally, there should be no difference between a local and a remote procedure call. However, the distribution of an application can result in a number of failures that would not occur with a centralized solution. Of all the possible failures that can occur, we will look at two particular types in detail: the loss of messages and the crash of a process. The different failures are shown in Figure 1.8.

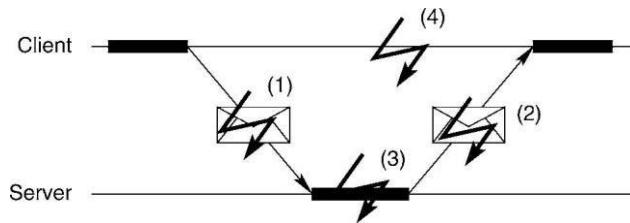


Fig. 1.8 Error situations with RPC.

Loss of request message (1): If a request message is lost, the client must retransmit the message after a timeout.

However, the client cannot differentiate between different types of failures. For example, if the result message is the one that is lost, a retransmission of the request message could result in the procedure being executed twice. The same problem occurs with long procedures when too short a timeout is selected.

Loss of result message (2): The procedure was executed on the server, but the result message for the client is lost.

The client retransmits the request after a timeout. If the server does not recognize what happened, it executes the procedure again. This can cause a problem with procedures that change the state of the server.

Server breakdown (3): If the server breaks down due to a failure, it has to be determined whether a partial execution of the procedure had already produced side effects in the state. For example, if the content of a database is modified during the procedure, it is not trivial to allow the execution to recover and continue in an ordered way after the crash of the server.

Client breakdown (4): A client process that breaks down during the execution of a remote procedure call is also referred to as an *orphaned invocation*. The question here is what the server does with the results or where it should send them.

Ideally, a remote procedure call should implement *exactly once* semantics—the invocation by a client will result in exactly one execution on the part of the server and also only delivers one result. However, it is not easy to achieve these semantics. Different applications have different requirements for quality of service in terms of failure detection and recovery. For instance, exactly once semantics are particularly desirable for bank transactions. However, repeated executions and numerous result messages would not be a problem in the case of a simple information terminal that only calls up data from a remote server without changing the state of the server (also referred to as *idempotent operations*). In this case, weak assurances of quality of service are sufficient in certain failure situations.

*Maybe* semantics provide no mechanism for lost messages or process break downs. Depending on the particular failure, the procedure can be carried out zero times or once on the server side. Consequently, the client receives at most one result. Maybe semantics essentially provide no guarantees. So long as no failures occur, the remote procedure call is properly carried out. These semantics are also referred to as *best-effort*.

*At-least-once* semantics guarantee that a remote procedure call will be executed on the server side at least once in the event of message loss. After a timeout, the client repeats the remote procedure call until it receives a response from the server. What can happen as a result, however, is that a procedure will be carried out several times on the server. It is also possible that a client will receive several responses due to the repeated executions. At-least-once semantics do not provide a confirmation if the server breaks down. At-least-once semantics are appropriate with idempotent procedures that do not cause state changes on the server and therefore can be executed more than once without any harm.

*At-most-once* semantics guarantee that the procedure will be executed at most once—both in the case of message loss and server breakdown. If the server does not break down, exactly one execution and exactly one result are even guaranteed. These semantics require a complex protocol with message buffering and numbering. Table 1.2 summarizes the failure semantics discussed along with their characteristics.

**Table 1.2 Failure semantics with RPC**

Failure semantics	Fault-free operation	Message loss	Server breakdown
Maybe	Execution: 1 Result: 1	Execution: 0/1 Result: 0	Execution: 0/1 Result: 0
At-least-once	Execution: 1 Result: 1	Execution: >=1 Result: >=1	Execution: >=0 Result: >=0
At-most-once	Execution: 1 Result: 1	Execution: 1 Result: 1	Execution: 0/1 Result: 0
Exactly once	Execution: 1 Result: 1	Execution: 1 Result: 1	Execution: 1 Result: 1

### 1.2.3 Group Communication

An underlying assumption intrinsic to RPC is that communication involves only two parties, the client and the server. Sometimes there are circumstances in which communication involves multiple processes, not just two. For example, consider a group of file servers cooperating to offer a single, fault-tolerant file.

In such a system, it might be desirable for a client to send a message to all the servers, to make sure that the request could be carried out even if one of them crashed. RPC cannot handle communication from one sender to many receivers, other than by performing separate RPCs with each one. In this section we will discuss alternative communication mechanisms in which a message can be sent to multiple receivers in one operation.

A *group* is a collection of processes that act together in some system or user-specified way. The key property that all groups have is that when a message is sent to the group itself, all members of the group receive it. It is a form of one-to-many communication (one sender, many receivers), and is contrasted with point-to-point communication.

Groups are dynamic. New groups can be created and old groups can be destroyed. A process can join a group or leave one. A process can be a member of several groups at the same time. Consequently, mechanisms are needed for managing groups and group membership.

Groups are roughly analogous to social organizations. A person might be a member of a book club, a tennis club, and an environmental organization. On a particular day, he might receive mailings (messages) announcing a new birthday cake cookbook from the book club, the annual Mother's Day tennis tournament from the tennis club, and the start of a campaign to save the Southern groundhog from the environmental organization. At any moment, he is free to leave any or all of these groups, and possibly join other groups.

The purpose of introducing groups is to allow processes to deal with collections of processes as a single abstraction. Thus a process can send a message to a group of servers without having to know how many there are or where they are, which may change from one call to the next.

How group communication is implemented depends to a large extent on the hardware. On some networks, it is possible to create a special network address (for example, indicated by setting one of the high-order bits to 1), to which multiple machines can listen. When a packet is sent to one of these addresses, it is automatically delivered to all machines listening to the address. This technique is called *multicasting*. Implementing groups using multicast is straightforward: just assign each group a different multicast address.

Networks that do not have multicasting sometimes still have broadcasting, which means that packets containing a certain address are delivered to all machines. Broadcasting can also be used to implement groups, but it is less efficient. Each machine receives each broadcast, so its software must check to see if the packet is intended for it. If not, the packet is discarded, but some time is wasted processing the interrupt. Nevertheless, it still takes only one packet to reach all the members of a group.

Finally, if neither multicasting nor broadcasting is available, group communication can still be implemented by having the sender transmit separate packets to each of the members of the group. For a group with  $n$  members,  $n$  packets are required, instead of one packet when either multicasting or broadcasting is used. Although less efficient, this implementation is still workable, especially if most groups are small. The sending of a message from a single sender to a single receiver is sometimes called uncasting (point-to-point transmission), to distinguish it from multicasting and broadcasting.

Group communication has many of the same design possibilities as regular message passing, such as buffered versus unbuffered, blocking versus nonblocking, and so forth. However, there are also a large number of additional choices that must be made because sending to a group is inherently different from sending to a single process. Furthermore, groups can be organized in various ways internally. They can also be addressed in novel ways not relevant in point-to-point communication.

The most important design issues and point out the various alternatives are:

- *Closed Groups versus Open Groups*. Some systems support closed groups, in which only the members of the group can send to the group. Outsiders cannot send messages to the group as a whole, although they may be able to send messages to individual members. In contrast, other systems support open groups, which do not have this property. When open groups are used, any process in the system can send to any group. A collection of processes working together to play a game of chess might form a closed group; they have their own goal and do not interact with the outside world. On the other hand, when the idea of groups is to support replicated servers, it is important that processes that are not members (clients) can send to the group.
- *Peer Groups versus Hierarchical Groups* In some groups, all the processes are equal. No one is boss and all decisions are made collectively. In other groups, some kind of hierarchy exists. For example, one process is the coordinator and all the others are workers. In this model, when a request for work is generated, either by an external client or by one of the workers, it is sent to the coordinator. The coordinator then decides which worker is best suited to carry it out, and forwards it there.
  1. The peer group is symmetric and has no single point of failure. If one of the processes crashes, the group simply becomes smaller, but can otherwise continue. A disadvantage is that decision making is more complicated. To decide anything, a vote has to be taken, incurring some delay and overhead.
  2. The hierarchical group has the opposite properties. Loss of the coordinator brings the entire group to a grinding halt, but as long as it is running, it can make decisions without bothering everyone else. For example, a hierarchical group might be appropriate for a parallel chess program. The coordinator takes the current board, generates all the legal moves from it, and farms them out to the workers for evaluation. During this evaluation, new boards are generated and sent back to the coordinator to have them evaluated. When a worker is idle, it asks the coordinator for a new board to work on. In this manner, the coordinator controls the search strategy and prunes the game tree (e.g. using the alpha-beta search method), but leaves the actual evaluation to the workers.
- *Group Membership* When group communication is present, some method is needed for creating and deleting groups, as well as for allowing processes to join and leave groups.
  1. One possible approach is to have a group server to which all these requests can be sent.
  2. The opposite approach is to manage group membership in a distributed way. In an open group, an outsider can send a message to all group members announcing its presence. In a closed group, something similar is needed (in effect, even closed groups have to be open with respect to joining). To leave a group, a member just sends a goodbye message to everyone.
- *Group Addressing*
  1. One way is to give each group a unique address, much like a process address. If the network supports multicast, the group address can be associated with a multicast address, so that every message sent to the group address can be multicast. In this way, the message will be sent to all those machines that need it, and no others. If the hardware supports broadcast but not multicast, the message can be broadcast. Finally, if neither multicast nor broadcast is supported, the kernel on the sending machine will have to have a list of machines that have processes belonging to the group.
  2. A second method of group addressing is to require the sender to provide an explicit list of all destinations.
  3. Group communication also allows a third, and quite novel method of addressing as well, which we will call predicate addressing. With this system, each message is sent to all members of the group (or possibly the entire system) using one of the methods described above, but with a new twist. Each message contains a predicate (Boolean expression) to be evaluated. The predicate can involve the receiver's machine number, its local variables, or other factors. If the predicate evaluates to TRUE, the message is accepted. If it evaluates to FALSE, the message is discarded. Using this scheme it is possible, for example, to send a message to only those machines that have at least 4M of free memory and which are willing to take on a new process.
- *Send and Receive Primitives* The library procedures that processes call to invoke group communication may be the same as for point-to-point communication or they may be different.

1. Suppose, for the moment, that we wish to merge the two forms of communication. To send a message, one of the parameters of send indicates the destination. If it is a process address, a single message is sent to that one process. If it is a group address (or a pointer to a list of destinations), a message is sent to all members of the group. A second parameter to send points to the message. The call can be buffered or unbuffered, blocking or nonblocking, reliable or not reliable, for both the point-to-point and group cases. Generally, these choices are made by the system designers and are fixed, rather than being selectable on a per message basis. Introducing group communication does not change this.
  2. Similarly, receive indicates a willingness to accept a message, and possibly blocks until one is available. If the two forms of communication are merged, receive completes when either a point-to-point message or a group message arrives. However, since these two forms of communication are frequently used for different purposes, some systems introduce new library procedures, say, group-send and group-receive, so a process can indicate whether it wants a point-to-point or a group message.
- **Atomicity** Atomicity is desirable because it makes programming distributed systems much easier. When any process sends a message to the group, it does not have to worry about what to do if some of them do not get it. For example, in a replicated distributed data base system, suppose that a process sends a message to all the data base machines to create a new record in the database, and later sends a second message to update it.
  - **Message Ordering**
    1. The best guarantee is to have all messages delivered instantaneously and in the order in which they were sent. All recipients get all messages in exactly the same order. This delivery pattern is something that programmers can understand and base their software on. We will call this *global time ordering*, since it delivers all messages in the exact order in which they were sent (conveniently ignoring the fact that there is no such thing as absolute global time).
    2. Absolute time ordering is not always easy to implement, so some systems offer various watered-down variations. One of these is *consistent time ordering*, in which if two messages, say A and B, are sent close together in time, the system picks one of them as being "first" and delivers it to all group members, followed by the other. It may happen that the one chosen as first was not really first, but since no one knows this, the argument goes, system behavior should not depend on it. In effect, messages are guaranteed to arrive at all group members in the same order, but that order may not be the real order in which they were sent.
  - **Overlapping Groups** The culprit here is that although there is a global time ordering within each group, there is not necessarily any coordination among multiple groups. Some systems support well-defined time ordering among overlapping groups and others do not. (If the groups are disjoint, the issue does not arise.) Implementing time ordering among different groups is frequently difficult to do, so the question arises as to whether it is worth it.
  - **Scalability**

#### **1.2.4 Problems**

1. What is meant by an open system? Why are some systems not open?
2. What is the difference between a connection-oriented and connectionless communication protocol?
3. If the communication primitives in a client-server system are nonblocking, a call to send will complete before the message has actually been sent. To reduce overhead, some systems do not copy the data to the kernel, but transmit it directly from user space. For such a system, devise two ways in which the sender can be told that the transmission has been completed and the buffer can be reused.
4. When buffered communication is used, a primitive is normally available for user processes to create mailboxes. In the text it was not specified whether this primitive must specify the size of the mailbox. Give an argument each way.
5. In all the examples in this chapter, a server can only listen to a single address. In practice, it is sometimes convenient for a server to listen to multiple addresses at the same time, for example, if the same process performs a set of closely related services that have been assigned separate addresses. Invent a scheme by which this goal can be accomplished.
6. One way to handle parameter conversion in RPC systems is to have each machine send parameters in its native representation, with the other one doing the translation, if need be. In the text it was suggested that the native system could be indicated by a code in the first byte. However, since locating the first byte in the first word is precisely the problem, can this work, or is the book wrong?

# 1.3 Synchronization in Distributed Systems

While communication is important, it is not the entire story. Closely related is how processes cooperate and synchronize with one another. For example, how are critical regions implemented in a distributed system, and how are resources allocated? In this chapter we will study these and other issues related to interprocess cooperation and synchronization in distributed systems.

In single CPU systems, critical regions, mutual exclusion, and other synchronization problems are generally solved using methods such as semaphores and monitors. These methods are not well suited to use in distributed systems because they invariably rely (implicitly) on the existence of shared memory. For example, two processes that are interacting using a semaphore must both be able to access the semaphore. If they are running on the same machine, they can share the semaphore by having it stored in the kernel, and execute system calls to access it. If, however, they are running on different machines, this method no longer works, and other techniques are needed. Even seemingly simple matters, such as determining whether event A happened before or after event B, require careful thought.

We will start out by looking at time and how it can be measured, because time plays a major role in some synchronization methods. Then we will look at mutual exclusion and election algorithms. After that we will study a high-level synchronization technique called atomic transactions. Finally, we will look at deadlock in distributed systems.

## 1.3.1 Clock Synchronization

Synchronization in distributed systems is more complicated than in centralized ones because the former have to use distributed algorithms. It is usually not possible (or desirable) to collect all the information about the system in one place, and then let some process examine it and make a decision as is done in the centralized case. In general, distributed algorithms have the following properties:

1. The relevant information is scattered among multiple machines.
2. Processes make decisions based only on local information.
3. A single point of failure in the system should be avoided.
4. No common clock or other precise global time source exists.

The first three points all say that it is unacceptable to collect all the information in a single place for processing. For example, to do resource allocation (assigning I/O devices in a deadlock-free way), it is generally not acceptable to send all the requests to a single manager process, which examines them all and grants or denies requests based on information in its tables. In a large system, such a solution puts a heavy burden on the manager.

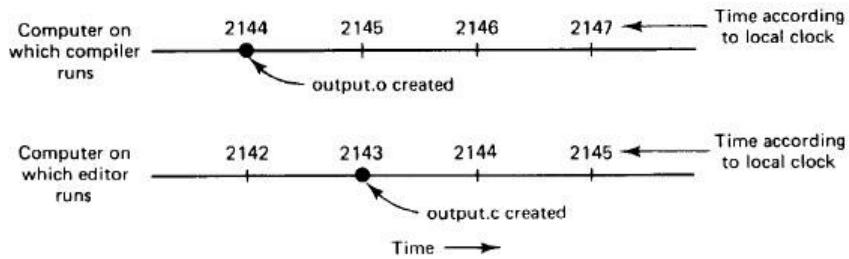
Furthermore, having a single point of failure like this makes the system unreliable. Ideally, a distributed system should be more reliable than the individual machines. If one goes down, the rest should be able to continue to function. Having the failure of one machine (e.g. the resource allocator) bring a large number of other machines (its customers) to a grinding halt is the last thing we want. Achieving synchronization without centralization requires doing things in a different way from traditional operating systems.

The last point in the list is also crucial. In a centralized system, time is unambiguous. When a process wants to know the time, it makes a system call and the kernel tells it. If process A asks for the time, and then a little later process B asks for the time, the value that B gets will be higher than (or possibly equal to) the value A got. It will certainly not be lower. In a distributed system, achieving agreement on time is not trivial.

Just think about the implications of the lack of global time on the Unix make program, as a single example. Normally, in Unix, large programs are split up into multiple source files, so that a change to one source file only requires one file to be recompiled, not all the files. If a program consists of 100 files, not having to recompile everything because one file has been changed greatly increases the speed at which programmers can work.

The way make normally works is simple. When the programmer has finished changing all the source files, he starts make, which examines the times at which all the source and object files were last modified. If the source file input.c has time 2151 and the corresponding object file has time 2150, make knows that input.c has been changed since input.o was created, and thus input.c must be recompiled. On the other hand, if output.c has time 2144 and output.o has time 2145, no compilation is needed here. Thus make goes through all the source files to find out which ones need to be recompiled and calls the compiler to recompile them.

Now imagine what could happen in a distributed system in which there is no global agreement on time. Suppose that has time 2144 as above, and shortly thereafter is modified but is assigned time 2143 because the clock on its machine is slightly slow, as shown in Fig. 1.9.



**Fig. 1.9. When each machine has its own clock, an event that occurred after another event may nevertheless be assigned an earlier time.**

Make will not call the compiler. The resulting executable binary program will then contain a mixture of object files from the old sources and the new sources. It will probably not work, and the programmer will go crazy trying to understand what is wrong with the code.

Since time is so basic to the way people think, and the effect of not having all the clocks synchronized can be so dramatic, as we have just seen, it is fitting that we begin our study of synchronization with the simple question: Is it possible to synchronize all the clocks in a distributed system?

### 1.3.1.1 Logical Clocks

Nearly all computers have a circuit for keeping track of time. Despite the widespread use of the word "clock" to refer to these devices, they are not actually clocks in the usual sense. Timer is perhaps a better word. A computer timer is usually a precisely machined quartz crystal. When kept under tension, quartz crystals oscillate at a well-defined frequency that depends on the kind of crystal, how it is cut, and the amount of tension. Associated with each crystal are two registers, a counter and a holding register. Each oscillation of the crystal decrements the counter by one. When the counter gets to zero, an interrupt is generated and the counter is reloaded from the holding register. In this way, it is possible to program a timer to generate an interrupt 60 times a second, or at any other desired frequency. Each interrupt is called one *clock tick*.

When the system is booted initially, it usually asks the operator to enter the date and time, which is then converted to the number of ticks after some known starting date and stored in memory. At every clock tick, the interrupt service procedure adds one to the time stored in memory. In this way, the (software) clock is kept up to date.

With a single computer and a single clock, it does not matter much if this clock is off by a small amount. Since all processes on the machine use the same clock, they will still be internally consistent. For example, if the file input.c has time 2151 and file input.o has time 2150, make will recompile the source file, even if the clock is off by 2 and the true times are 2153 and 2152, respectively. All that really matters are the relative times.

As soon as multiple CPUs are introduced, each with its own clock, the situation changes. Although the frequency at which a crystal oscillator runs is usually fairly stable, it is impossible to guarantee that the crystals in different computers all run at exactly the same frequency. In practice, when a system has n computers, all n crystals will run at slightly different rates, causing the (software) clocks gradually to get out of sync and give different values when read out. This difference in time values is called *clock skew*. As a consequence of this clock skew, programs that expect the time associated with a file, object, process, or message to be correct and independent of the machine on which it was generated (i.e. which clock it used) can fail, as we saw in the make example above.

This brings us back to our original question, whether it is possible to synchronize all the clocks to produce a single, unambiguous time standard. In a classic paper, Lamport (1978) showed that clock synchronization is possible and presented an algorithm for achieving it.

Lamport pointed out that clock synchronization need not be absolute. If two processes do not interact, it is not necessary that their clocks be synchronized because the lack of synchronization would not be observable and thus could not cause problems. Furthermore, he pointed out that what usually matters is not that all processes agree on exactly what time it is, but rather, that they agree on the order in which events occur. In the make example above, what counts is whether input.c is older or newer than input.o not their absolute creation times.

For many purposes, it is sufficient that all machines agree on the same time. It is not essential that this time also agree with the real time as announced on the radio every hour. For running make, for example, it is adequate that all machines agree that it is 10:00 even if it is really 10:02. Thus for a certain class of algorithms, it is the internal consistency of the clocks that matters, not whether they are particularly close to the real time. For these algorithms, it is conventional to speak of the clocks as *logical clocks*.

When the additional constraint is present that the clocks must not only be the same, but also must not deviate from the real time by more than a certain amount, the clocks are called *physical clocks*. In this section we will discuss Lamport's algorithm, which synchronizes logical clocks. In the following sections we will introduce the concept of physical time and show how physical clocks can be synchronized.

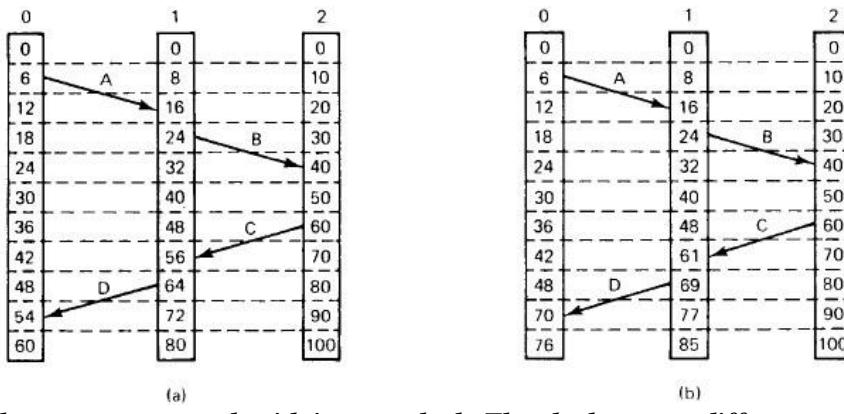
To synchronize logical clocks, Lamport defined a relation called *happens-before*. The expression  $a \rightarrow b$  is read " $a$  happens before  $b$ " and means that all processes agree that first event  $a$  occurs, then afterward, event  $b$  occurs. The happens-before relation can be observed directly in two situations:

1. If  $a$  and  $b$  are events in the same process, and  $a$  occurs before  $b$  then  $a \rightarrow b$  is true.

2. If  $a$  is the event of a message being sent by one process, and  $b$  is the event of the message being received by another process, then  $a \rightarrow b$  is also true. A message cannot be received before it is sent, or even at the same time it is sent, since it takes a finite amount of time to arrive.

Happens-before is a transitive relation, so if  $a \rightarrow b$  and  $b \rightarrow c$ , then  $a \rightarrow c$ . If two events,  $x$  and  $y$ , happen in different processes that do not exchange messages (not even indirectly via third parties), then  $x \rightarrow y$  is not true, but neither is  $y \rightarrow x$ . These events are said to be *concurrent*, which simply means that nothing can be said (or need be said) about when they happened or which is first.

What we need is a way of measuring time such that for every event,  $a$ , we can assign it a time value  $C(a)$  on which all processes agree. These time values must have the property that if  $a \rightarrow b$ , then  $C(a) < C(b)$ . To rephrase the conditions we stated earlier, if  $a$  and  $b$  are two events within the same process and  $a$  occurs before  $b$ , then  $C(a) < C(b)$ . Similarly, if  $a$  is the sending of a message by one process and  $b$  is the reception of that message by another process, then  $C(a)$  and  $C(b)$  must be assigned in such a way that everyone agrees on the values of  $C(a)$  and  $C(b)$  with  $C(a) < C(b)$ . In addition, the clock time,  $C$ , must always go forward (increasing), never backward (decreasing). Corrections to time can be made by adding a positive value, never by subtracting one.



**Fig. 1.10. (a) Three processes, each with its own clock. The clocks run at different rates. (b) Lamport's algorithm corrects the clocks.**

Now let us look at the algorithm Lamport proposed for assigning times to events. Consider the three processes depicted in Fig. 1.10. The processes run on different machines, each with its own clock, running at its own speed. As can be seen from the figure, when the clock has ticked 6 times in process 0, it has ticked 8 times in process 1 and 10 times in process 2. Each clock runs at a constant rate, but the rates are different due to differences in the crystals. At time 6, process 0 sends message A to process 1. How long this message takes to arrive depends on whose clock you believe. In any event, the clock in process 1 reads 16 when it arrives. If the message carries the starting time, 6, in it, process 1 will conclude that it took 10 ticks to make the journey. This value is certainly possible. According to this reasoning, message B from 1 to 2 takes 16 ticks, again a plausible value. Message from 2 to 1 leaves at 60 and arrives at 56. Similarly, message D from 1 to 0 leaves at 64 and arrives at 54. These values are clearly impossible. It is this situation that must be prevented. Lamport's solution follows directly from the happened-before relation. Since C left at 60, it must arrive at 61 or later. Therefore, each message carries the sending time, according to the sender's clock. When a message arrives and the receiver's clock shows a value prior

to the time the message was sent, the receiver fast forwards its clock to be one more than the sending time. In Fig. 1.10(b) we see that C now arrives at 61. Similarly, D arrives at 70.

With one small addition, this algorithm meets our requirements for global time. The addition is that between every two events, the clock must tick at least once. If a process sends or receives two messages in quick succession, it must advance its clock by (at least) one tick in between them.

In some situations, an additional requirement is desirable: no two events ever occur at exactly the same time. To achieve this goal, we can attach the number of the process in which the event occurs to the low-order end of the time, separated by a decimal point. Thus if events happen in processes 1 and 2, both with time 40, the former becomes 40.1 and the latter becomes 40.2.

Using this method, we now have a way to assign time to all events in a distributed system subject to the following conditions:

1. If  $a$  happens before  $b$  in the same process,  $C(a) < C(b)$ .
2. If  $a$  and  $b$  represent the sending and receiving of a message,  $C(a) < C(b)$ .
3. For all distinct events  $a$  and  $b$ ,  $C(a)$  is not equal with  $C(b)$ .

This algorithm gives us a way to provide a total ordering of all events in the system. Many other distributed algorithms need such an ordering to avoid ambiguities, so the algorithm is widely cited in the literature.

### 1.3.1.2 Physical Clocks

Although Lamport's algorithm gives an unambiguous event ordering, the time values assigned to events are not necessarily close to the actual times at which they occur. In some systems real-time systems, the actual clock time is important. For these systems external physical clocks are required. For reasons of efficiency and redundancy, multiple physical clocks are generally considered desirable, which yields two problems: (1) How do we synchronize them with real-world clocks, and (2) How do we synchronize the clocks with each other?

Before answering these questions, let us digress slightly to see how time is actually measured. It is not nearly as simple as one might think, especially when high accuracy is required. Since the invention of mechanical clocks in the 17th century, time has been measured astronomically. Every day, the sun appears to rise on the eastern horizon, climbs to a maximum height in the sky, and sinks in the west. The event of the sun's reaching its highest apparent point in the sky is called the transit of the sun. This event occurs at about noon each day. The interval between two consecutive transits of the sun is called the solar day. Since there are 24 hours in a day, each containing 3600 seconds, the *solar second* is defined as exactly 1186400th of a solar day.

In the 1940s it was established that the period of the earth's rotation is not constant. The earth is slowing down due to tidal friction and atmospheric drag. Based on studies of growth patterns in ancient coral, geologists now believe that 300 million years ago there were about 400 days per year. The length of the year, that is, the time for one trip around the sun, is not thought to have changed; the day has simply become longer. In addition to this long-term trend, short-term variations in the length of the day also occur, probably caused by turbulence deep in the earth's core of molten iron. These revelations led astronomers to compute the length of the day by measuring a large number of days and taking the average before dividing by 86,400. The resulting quantity was called the mean *solar second*.

With the invention of the atomic clock in 1948, it became possible to measure time much more accurately, and independent of the wiggling and wobbling of the earth, by counting transitions of the cesium 133 atom. The physicists took over the job of timekeeping from the astronomers, and defined the second to be the time it takes the cesium 133 atom to make exactly 1,770 transitions. The choice of 9,192,631,770 was made to make the atomic second equal to the mean solar second in the year of its introduction. Currently, about 50 laboratories around the world have cesium 133 clocks. Periodically, each laboratory tells the Bureau International de (BIH) in Paris how many times its clock has ticked. The BIH averages these to produce *International Atomic Time*, which is abbreviated TAI. Thus TAI is just the mean number of ticks of the cesium 133 clocks since midnight on Jan. 1, 1958 (the beginning of time) divided by 1,770.

Although TAI is highly stable and available to anyone who wants to go to the trouble of buying a cesium clock, there is a serious problem with it; 86,400 TAI seconds is now about 3 msec less than a mean solar day (because the mean solar day is getting longer all the time). Using TAI for keeping time would mean that over the course of the years, noon would get earlier and earlier, until it would eventually occur in the wee hours of the morning. People might notice this and we could have the same kind of situation as occurred in 1582 when Pope Gregory

XIII decreed that 10 days be omitted from the calendar. This event caused riots in the streets because landlords demanded a full month's rent and bankers a full month's interest, while employers refused to pay workers for the 10 days they did not work, to mention only a few of the conflicts. The Protestant countries, as a matter of principle, refused to have anything to do with papal decrees and did not accept the Gregorian calendar for 170 years.

BIH solves the problem by introducing *leap seconds* whenever the discrepancy between TAI and solar time grows to 800 msec. This correction gives rise to a time system based on constant TAI seconds but which stays in phase with the apparent motion of the sun. It is called *Universal Coordinated Time*, but is abbreviated as UTC. UTC is the basis of all modern civil timekeeping. It has essentially replaced the old standard, Greenwich Mean Time, which is astronomical time.

Most electric power companies base the timing of their 60-Hz or 50-Hz clocks on UTC, so when BIH announces a leap second, the power companies raise their frequency to 61 Hz or 51 Hz for 60 or 50 sec, to advance all the clocks in their distribution area. Since 1 sec is a noticeable interval for a computer, an operating system that needs to keep accurate time over a period of years must have special software to account for leap seconds as they are announced (unless they use the power line for time, which is usually too crude). The total number of leap seconds introduced into UTC so far is about 30.

To provide UTC to people who need precise time, the National Institute of Standard Time (NIST) operates a shortwave radio station with call letters WWV from Fort Collins, Colorado. WWV broadcasts a short pulse at the start of each UTC second. The accuracy of WWV itself is about 1 msec, but due to random atmospheric fluctuations that can affect the length of the signal path, in practice the accuracy is no better than 10 msec. In England, the station MSF, operating from Rugby, Warwickshire, provides a service, as do stations in several other countries.

Several earth satellites also offer a UTC service. The Geostationary Environment Operational Satellite can provide UTC accurately to 0.5 msec, and some other satellites do even better. Using either shortwave radio or satellite services requires an accurate knowledge of the relative position of the sender and receiver, in order to compensate for the signal propagation delay.

### 1.3.1.3 Clock Synchronization Algorithms

If one machine has a UTC receiver, the goal becomes keeping all the other machines synchronized to it. If no machines have UTC receivers, each machine keeps track of its own time, and the goal is to keep all the machines together as well as possible. Many algorithms have been proposed for doing this synchronization.

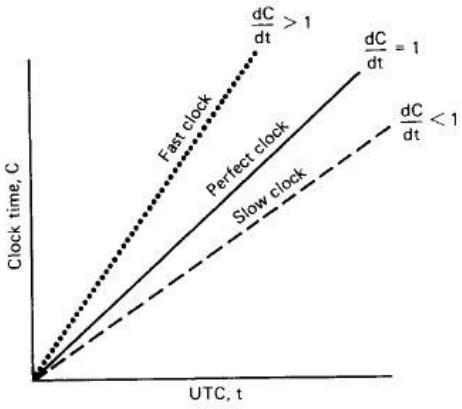
All the algorithms have the same underlying model of the system, which we will now describe. Each machine is assumed to have a timer that causes an interrupt  $H$  times a second. When this timer goes off, the interrupt handler adds 1 to a software clock that keeps track of the number of ticks (interrupts) since some agreed-upon time in the past. Let us call the value of this clock  $C$ . More specifically, when the UTC time is  $t$ , the value of the clock on machine  $p$  is  $C_p(t)$ . In a perfect world, we would have  $C_p(t) = t$  for all  $p$  and all  $t$ . In other words,  $dC/dt$  ideally should be 1.

Real timers do not interrupt exactly  $H$  times a second. Theoretically, a timer with  $H = 60$  should generate 216,000 ticks per hour. In practice, the relative error obtainable with modern timer chips is about meaning that a particular machine can get a value in the range 215,998 to 216,002 ticks per hour. More precisely, if there exists some constant such that

$$1-r \leq dC/dt \leq 1+r$$

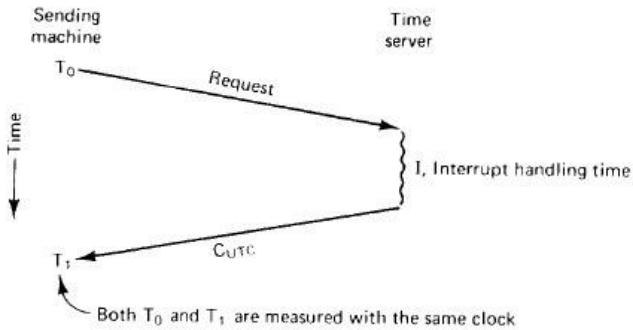
the timer can be said to be working within its specification. The constant  $r$  is specified by the manufacturer and is known as the maximum *drift rate*. Slow, perfect, and fast clocks are shown in Fig. 1.11.

If two clocks are drifting from UTC in the opposite direction, at a time  $dt$  after they were synchronized, they may be as much as  $2r dt$ . If the operating system designers want to guarantee that no two clocks ever differ by more than  $d$ , clocks must be resynchronized (in software) at least every  $d/2r$  seconds. The various algorithms differ in precisely how this resynchronization is done.



**Fig. 1.11. Not all clocks tick precisely at the correct rate.**

**Cristian's Algorithm.** Let us start with an algorithm that is well suited to systems in which one machine has a UTC receiver and the goal is to have all the other machines stay synchronized with it. Let us call the machine with the WWV receiver a *time server*. Periodically, certainly no more than every  $d/2r$  seconds, each machine sends a message to the time server asking it for the current time. That machine responds  $C_{\text{UTC}}$  as fast as it can with a message containing its current time, as shown in Fig. 1.12.



**Fig. 1.12. Getting the current time from a time server.**

As a first approximation, when the sender gets the reply, it can just set its clock to  $C_{\text{UTC}}$ . However, this algorithm has two problems, one major and one minor. The major problem is that time must never run backward. If the sender's clock is fast, will be smaller than the sender's current value of  $C$ . Just taking over could cause serious problems, such as an object file compiled just after the clock change having a time earlier than the source which was modified just before the clock change.

Such a change must be introduced gradually. One way is as follows. Suppose that the timer is set to generate 100 interrupts per second. Normally, each interrupt would add 10 msec to the time. When slowing down, the interrupt routine adds only 9 msec each time, until the correction has been made. Similarly, the clock can be advanced gradually by adding 11 msec at each interrupt instead of jumping it forward all at once.

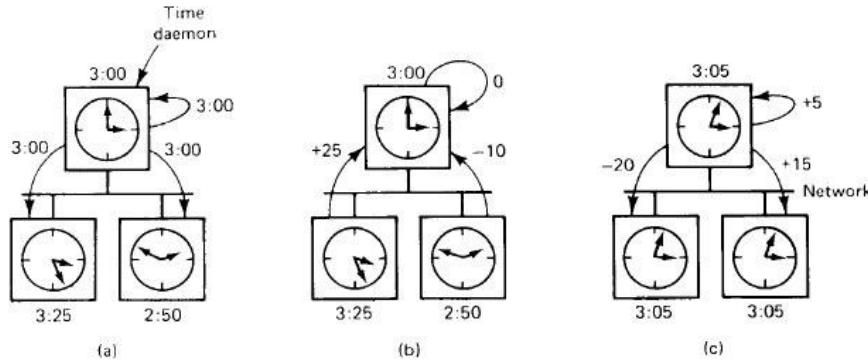
The minor problem is that it takes a nonzero amount of time for the time server's reply to get back to the sender. Worse yet, this delay may be large and vary with the network load. Cristian's way of dealing with it is to attempt to measure it. It is simple enough for the sender to record accurately the interval between sending the request to the time server and the arrival of the reply. Both the starting time,  $T_0$  and the ending time,  $T_1$ , are measured using the same clock, so the interval will be relatively accurate, even if the sender's clock is off from UTC by a substantial amount.

In the absence of any other the best estimate of the message propagation time is  $(T_1 - T_0)/2$ . When the reply comes in, the value in the message can be increased by this amount to give an estimate of the server's current time. If the theoretical minimum propagation time is known, other properties of the time estimate can be calculated.

This estimate can be improved if it is known approximately how long it takes the time server to handle the interrupt and process the incoming message. Let us call the interrupt handling time  $I$ . Then the amount of the interval from  $T_0$  to  $T_1$  that was devoted to message propagation is  $T_1 - T_0 - I$  so the best estimate of the one-way propagation time is half this. Systems do exist in which messages from A to B systematically take a different route than messages from B to A, and thus have a different propagation time.

To improve the accuracy, Cristian suggested making not one measurement, but a series of them. Any measurements in which  $T_1 - T_0$  exceeds some threshold value are discarded as being victims of network congestion and thus unreliable. The estimates derived from the remaining probes can then be averaged to get a better value. Alternatively, the message that came back fastest can be taken to be the most accurate since it presumably encountered the least traffic underway and thus is the most representative of the pure propagation time.

**The Berkeley Algorithm.** In Cristian's algorithm, the time server is passive. Other machines ask it for the time periodically. All it does is respond to their queries. In Berkeley algorithm exactly the opposite approach is taken. Here the time server (actually, a time daemon) is active, polling every machine periodically to ask what time it is there. Based on the answers, it computes an average time and tells all the other machines to advance their clocks to the new time or slow their clocks down until some specified reduction has been achieved. This method is suitable for a system in which no machine has a WWV receiver. The time daemon's time must be set manually by the operator periodically. The method is illustrated in Fig. 1.13.



**Fig. 1.13. (a)** The time daemon asks all the other machines for their clock values. **(b)** The machines answer. **(c)** The time daemon tells everyone how to adjust their clock.

In Fig. 1.13(a), at 3:00, the time daemon tells the other machines its time and asks for theirs. In Fig. 1.13(b), they respond with how far ahead or behind the time daemon they are. Armed with these numbers, the time daemon computes the average and tells each machine how to adjust its clock (see Fig. 1.13(c)).

**Averaging Algorithms.** Both of the methods described above are highly centralized, with the usual disadvantages. Decentralized algorithms are also known. One class of decentralized clock synchronization algorithms works by dividing time into fixed-length resynchronization intervals. The  $i$ th interval starts at  $T_0+iR$  and runs until  $T_0+(i+1)R$  where  $T_0$  is an agreed upon moment in the past, and  $R$  is a system parameter. At the beginning of each interval, every machine broadcasts the current time according to its clock. Because the clocks on different machines do not run at exactly the same speed, these broadcasts will not happen precisely simultaneously.

After a machine broadcasts its time, it starts a local timer to collect all other broadcasts that arrive during some interval  $S$ . When all the broadcasts arrive, an algorithm is run to compute a new time from them. The simplest algorithm is just to average the values from all the other machines. A slight variation on this theme is first to discard the  $m$  highest and  $m$  lowest values, and average the rest. Discarding the extreme values can be regarded as self defense against up to  $m$  faulty clocks sending out nonsense.

Another variation is to try to correct each message by adding to it an estimate of the propagation time from the source. This estimate can be made from the known topology of the network, or by timing how long it takes for probe messages to be echoed.

### 1.3.1.4 Multiple External Time Sources

For systems in which extremely accurate synchronization with UTC is required, it is possible to equip the system with multiple receivers for WWV, GEOS, or other UTC sources. However, due to inherent inaccuracy in the time source itself as well as fluctuations in the signal path, the best the operating system can do is establish a range (time interval) in which UTC falls. In general, the various time sources will produce different ranges, which requires the machines attached to them to come to agreement.

To reach this agreement, each processor with a UTC source can broadcast its range periodically, say, at the precise start of each UTC minute. None of the processors will get the time packets instantaneously. Worse yet, the delay

between transmission and reception depends on the cable distance and number of gateways that the packets have to traverse, which is different for each (UTC source, processor) pair. Other factors can also play a role, such as delays due to collisions when multiple machines try to transmit on an Ethernet at the same instant. Furthermore, if a processor is busy handling a previous packet, it may not even look at the time packet for a considerable number of milliseconds, introducing additional uncertainty into the time.

### 1.3.1.5 Use of Synchronized Clocks

**At-Most-Once Message Delivery.** Our first example concerns how to enforce at-most-once message delivery to a server, even in the face of crashes. The traditional approach is for each message to bear a unique message number, and have each server store all the numbers of the messages it has seen so it can detect new messages from retransmissions. The problem with this algorithm is that if a server crashes and reboots, it loses its table of message numbers. Also, for how long should message numbers be saved?

Using time, the algorithm can be modified as follows. Now, every message carries a connection identifier (chosen by the sender) and a timestamp. For each connection, the server records in a table the most recent timestamp it has seen. If any incoming message for a connection is lower than the timestamp stored for that connection, the message is rejected as a duplicate. To make it possible to remove old timestamps, each server continuously maintains a global variable  $G = \text{CurrentTime} - \text{MaxLifetime} - \text{MaxClockSkew}$  where `MaxLifetime` is the maximum time a message can live and `MaxClockSkew` is how far from UTC the clock might be at worst. Any timestamp older than  $G$  can safely be removed from the table because all messages that old have died out already. If an incoming message has an unknown connection identifier, it is accepted if its timestamp is more recent than  $G$  and rejected if its timestamp is older than  $G$  because anything that old surely is a duplicate. In effect,  $G$  is a summary of the message numbers of all old messages. Every  $dT$ , the current time is written to disk.

**Clock-Based Cache Consistency.** Our second example concerns cache consistency in a distributed file system. For performance reasons, it is desirable for clients to be able to cache files locally. However, caching introduces potential inconsistency if two clients modify the same file at the same time. The usual solution is to distinguish between caching a file for reading and caching a file for writing. The disadvantage of this scheme is that if a client has a file cached for reading, before another client can get a copy for writing, the server has to first ask the reading client to invalidate its copy, even if the copy was made hours ago. This extra overhead can be eliminated using synchronized clocks.

The basic idea is that when a client wants a file, it is given a lease on it that specifies how long the copy is valid. When the lease is about to expire, the client can ask for it to be renewed. If a lease expires, the cached copy may no longer be used. In this way when a client needs to read a file once, it can ask for it. When the lease expires, it just times out; there is no need to explicitly send a message telling the server that it has been purged from the cache. If a lease has expired and the file (still cached) is needed again shortly thereafter, the client can ask the server if the copy it has (identified by a time-stamp) is still the current one. If so, a new lease is generated, but the file need not be retransmitted.

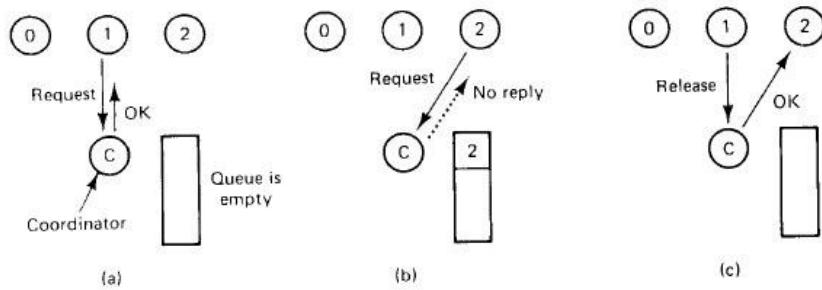
In addition to these two algorithms, synchronized clocks can be used to time out tickets used in distributed system authentication, and handle commitment in atomic transactions.

### 1.3.2 Mutual Exclusion

Systems involving multiple processes are often most easily programmed using critical regions. When a process has to read or update certain shared data structures, it enters a critical region to achieve mutual exclusion and ensure that no other process will use the shared data structures at the same time. In single-processor systems, critical regions are protected using semaphores, monitors, and similar constructs. We will now look at a few examples of how critical regions and mutual exclusion can be implemented in distributed systems.

#### 1.3.2.1 A Centralized Algorithm

The most straightforward way to achieve mutual exclusion in a distributed system is to simulate how it is done in a one-processor system. One process is elected as the coordinator (e.g. the one running on the machine with the highest network address). Whenever a process wants to enter a critical region, it sends a request message to the coordinator stating which critical region it wants to enter and asking for permission. If no other process is currently in that critical region, the coordinator sends back a reply granting permission, as shown in Fig. 1.14. When the reply arrives, the requesting process enters the critical region.



**Fig. 1.14.** (a) *Process 1 asks the coordinator for permission to enter a critical region. Permission is granted.* (b) *Process 2 then asks permission to enter the same critical region. The coordinator does not reply.* (c) *When process 1 exits the critical region, it tells the coordinator, which then replies to 2.*

Now suppose that another process, 2 in Fig. 1.14(b) asks for permission to enter the same critical region. The coordinator knows that a different process is already in the critical region, so it cannot grant permission. The exact method used to deny permission is system dependent. In Fig. 1.14(b) the coordinator just refrains from replying, thus blocking process 2, which is waiting for a reply. Alternatively, it could send a reply saying "permission denied." Either way, it queues the request from 2 for the time being.

When process 1 exits the critical region, it sends a message to the coordinator releasing its exclusive access, as shown in Fig. 1.14(c). The coordinator takes the first item off the queue of deferred requests and sends that process a grant message. If the process was still blocked (i.e. this is the first message to it), it unblocks and enters the critical region. If an explicit message has already been sent denying permission, the process will have to poll for incoming traffic, or block later. Either way, when it sees the grant, it can enter the critical region.

It is easy to see that the algorithm guarantees mutual exclusion: the coordinator only lets one process at a time into each critical region. It is also fair, since requests are granted in the order in which they are received. No process ever waits forever (no starvation). The scheme is easy to implement, too, and requires only three messages per use of a critical region (request, grant, release). It can also be used for more general resource allocation rather than just managing critical regions.

The centralized approach also has shortcomings. The coordinator is a single point of failure, so if it crashes, the entire system may go down. If processes normally block after making a request, they cannot distinguish a dead coordinator from "permission denied" since in both cases no message comes back. In addition, in a large system, a single coordinator can become a performance bottleneck.

### 1.3.2.2 A Distributed Algorithm

Ricart and Agrawala's algorithm requires that there be a total ordering of all events in the system. That is, for any pair of events, such as messages, it must be unambiguous which one happened first. Lamport's algorithm is one way to achieve this ordering and can be used to provide time-stamps for distributed mutual exclusion.

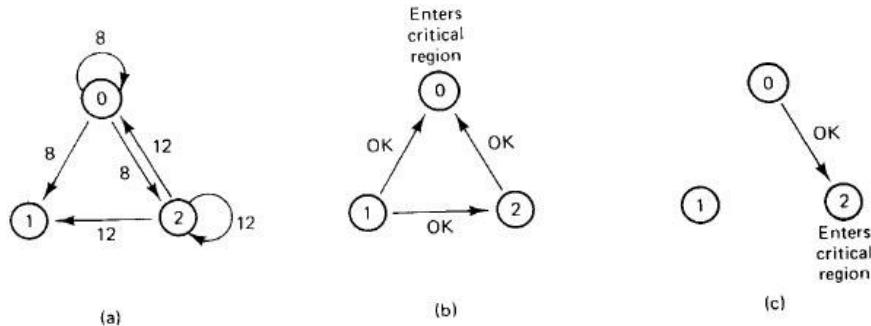
When a process wants to enter a critical region, it builds a message containing the name of the critical region it wants to enter, its process number, and the current time. It then sends the message to all other processes, conceptually including itself. The sending of messages is assumed to be reliable; that is, every message is acknowledged. Reliable group communication if available, can be used instead of individual messages.

When a process receives a request message from another process, the action it takes depends on its state with respect to the critical region named in the message. Three cases have to be distinguished:

1. If the receiver is not in the critical region and does not want to enter it, it sends back an OK message to the sender.
2. If the receiver is already in the critical region, it does not reply. Instead, it queues the request.
3. If the receiver wants to enter the critical region but has not yet done so, it compares the timestamp in the incoming message with the one contained in the message that it has sent everyone. The lowest one wins. If the incoming message is lower, the receiver sends back an OK message. If its own message has a lower timestamp, the receiver queues the incoming request and sends nothing.

After sending out requests asking permission to enter a critical region, a process sits back and waits until everyone else has given permission. As soon as all the permissions are in, it may enter the critical region. When it exits the critical region, it sends OK messages to all processes on its queue and deletes them all from the queue.

Let us try to understand why the algorithm works. If there is no conflict, it clearly works. However, suppose that two processes try to enter the same critical region simultaneously, as shown in Fig. 1.15(a).



**Fig. 1.15.** (a) Two processes want to enter the same critical region at the same moment. (b) Process 0 has the lowest timestamp, so it wins. (c) When process 0 is done, it sends an OK, so 2 can now enter the critical region.

Process 0 sends everyone a request with timestamp 8, while at the same time, process 2 sends everyone a request with timestamp 12. Process 1 is not interested in entering the critical region, so it sends OK to both senders. Processes 0 and 2 both see the conflict and compare timestamps. Process 2 sees that it has lost, so it grants permission to by sending OK. Process 0 now queues the request from 2 for later processing and enters the critical region, as shown in Fig. 1.15(b). When it is finished, it removes the request from 2 from its queue and sends an OK message to process 2, allowing the latter to enter its critical region, as shown in Fig. 1.15(c). The algorithm works because in the case of a conflict, the lowest timestamp wins and everyone agrees on the ordering of the timestamps. Note that the situation in Fig. 1.15 would have been essentially different if process 2 had sent its message earlier in time so that process 0 had gotten it and granted permission before making its own request. In this case, 2 would have noticed that it itself was in a critical region at the time of the request, and queued it instead of sending a reply.

Mutual exclusion is guaranteed without deadlock or starvation. The number of messages required per entry is now  $2(n - 1)$  where the total number of processes in the system is  $n$ . Best of all, no single point of failure exists.

Unfortunately, the single point of failure has been replaced by  $n$  points of failure. If any process crashes, it will fail to respond to requests. This silence will be interpreted (incorrectly) as denial of permission, thus blocking all subsequent attempts by all processes to enter all critical regions. Since the probability of one of the  $n$  processes failing is  $n$  times as large as a single coordinator failing, we have managed to replace a poor algorithm with one that is times worse and requires much more network traffic to boot.

The algorithm can be patched up by the same trick that we proposed earlier. When a request comes in, the receiver always sends a reply, either granting or denying permission. Whenever either a request or a reply is lost, the sender times out and keeps trying until either a reply comes back or the sender concludes that the destination is dead. After a request is denied, the sender should block waiting for a subsequent OK message.

Another problem with this algorithm is that either a group communication primitive must be used, or each process must maintain the group membership list itself, including processes entering the group, leaving the group, and crashing. The method works best with small groups of processes that never change their group memberships.

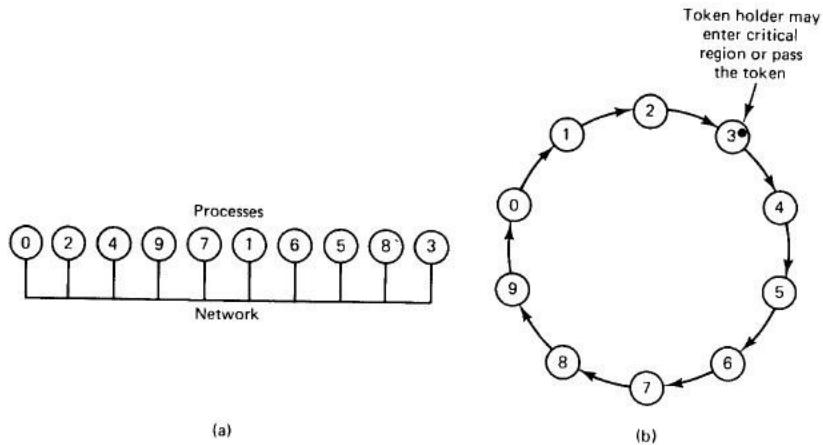
Finally, recall that one of the problems with the centralized algorithm is that making it handle all requests can lead to a bottleneck. In the distributed algorithm, all processes are involved in all decisions concerning entry into critical regions. If one process is unable to handle the load, it is unlikely that forcing everyone to do exactly the same thing in parallel is going to help much.

Various minor improvements are possible to this algorithm. For example, getting permission from everyone to enter a critical region is really overkill. All that is needed is a method to prevent two processes from entering the critical region at the same time. The algorithm can be modified to allow a process to enter a critical region when it has collected permission from a simple majority of the other processes, rather than from all of them. Of course, in this variation, after a process has granted permission to one process to enter a critical region, it cannot grant the same permission to another process until the first one has released that permission.

Nevertheless, this algorithm is slower, more complicated, more expensive, and less robust than the original centralized one.

### 1.3.2.3 A Token Ring Algorithm

A completely different approach to achieving mutual exclusion in a distributed system is illustrated in Fig. DOS.8 we have a bus network, as shown in Fig. 1.16(a) (e.g. Ethernet), with no inherent ordering of the processes. In software, a logical ring is constructed in which each process is assigned a position in the ring, as shown in Fig. 1.16(b). The ring positions may be allocated in numerical order of network addresses or some other means. It does not matter what the ordering is. All that matters is that each process knows who is next in line after itself.



**Fig. 1.16. (a) An unordered group of processes on a network. (b) A logical ring constructed in software.**

When the ring is initialized, process 0 is given a token. The token circulates around the ring. It is passed from process  $k$  to process  $k+1$  (modulo the ring size) in point-to-point messages. When a process acquires the token from its neighbor, it checks to see if it is attempting to enter a critical region. If so, the process enters the region, does all the work it needs to, and leaves the region. After it has exited, it passes the token along the ring. It is not permitted to enter a second critical region using the same token.

If a process is handed the token by its neighbor and is not interested in entering a critical region, it just passes it along. As a consequence, when no processes want to enter any critical regions, the token just circulates at high speed around the ring.

The correctness of this algorithm is evident. Only one process has the token at any instant, so only one process can be in a critical region. Since the token circulates among the processes in a well-defined order, starvation cannot occur. Once a process decides it wants to enter a critical region, at worst it will have to wait for every other process to enter and leave one critical region.

As usual, this algorithm has problems too. If the token is ever lost, it must be regenerated. In fact, detecting that it is lost is difficult, since the amount of time between successive appearances of the token on the network is unbounded. The fact that the token has not been spotted for an hour does not mean that it has been lost; somebody may still be using it.

The algorithm also runs into trouble if a process crashes, but recovery is easier than in the other cases. If we require a process receiving the token to acknowledge receipt, a dead process will be detected when its neighbor tries to give it the token and fails. At that point the dead process can be removed from the group, and the token holder can throw the token over the head of the dead process to the next member down the line, or the one after that, if necessary. Of course, doing so requires that everyone maintains the current ring configuration.

### 1.3.2.4 A Comparison of the Three Algorithms

A brief comparison of the three mutual exclusion algorithms we have looked at is instructive. We have listed the algorithms and three key properties: the number of messages required for a process to enter and exit a critical region, the delay before entry can occur (assuming messages are passed sequentially over a LAN), and some problems associated with each algorithm.

Algorithm	Messages	Delay before entry	Problems
Centralized	3	2	Coordinator crash
Distributed	$2(n-1)$	$2(n-1)$	Crash of any process
Token ring	1 to infinity	0 to $n - 1$	Lost token, process crash

The centralized algorithm is simplest and also most efficient. It requires only three messages to enter and leave a critical region: a request and a grant to enter, and a release to exit. The distributed algorithm requires  $n - 1$  request messages, one to each of the other processes, and an additional  $n - 1$  grant messages, for a total of  $2(n - 1)$ . With the token ring algorithm, the number is variable. If every process constantly wants to enter a critical region, then each token pass will result in one entry and exit, for an average of one message per critical region entered. At the other extreme, the token may sometimes circulate for hours without anyone being interested in it. In this case, the number of messages per entry into a critical region is unbounded.

The delay from the moment a process needs to enter a critical region until its actual entry also varies for the three algorithms. When critical regions are short and rarely used, the dominant factor in the delay is the actual mechanism for entering a critical region. When they are long and frequently used, the dominant factor is waiting for everyone else to take their turn. It takes only two message times to enter a critical region in the centralized case, but  $2(n - 1)$  message times in the distributed case, assuming that the network can handle only one message at a time. For the token ring, the time varies from 0 (token just arrived) to  $n - 1$  (token just departed).

Finally, all three algorithms suffer badly in the event of crashes. Special measures and additional complexity must be introduced to avoid having a crash bring down the entire system. It is slightly ironic that the distributed algorithms are even more sensitive to crashes than the centralized one. In a fault-tolerant system, none of these would be suitable, but if crashes are very infrequent, they are all acceptable.

### **1.3.3 Election Algorithms**

Many distributed algorithms require one process to act as coordinator, initiator, sequencer, or otherwise perform some special role. We have already seen several examples, such as the coordinator in the centralized mutual exclusion algorithm. In general, it does not matter which process takes on this special responsibility, but one of them has to do it. In this section we will look at algorithms for electing a coordinator.

If all processes are exactly the same, with no distinguishing characteristics, there is no way to select one of them to be special. Consequently, we will assume that each process has a unique number, for example its network address (for simplicity, we will assume one process per machine). In general, election algorithms attempt to locate the process with the highest process number and designate it as coordinator. The algorithms differ in the way they do the location.

Furthermore, we also assume that every process knows the process number of every other process. What the processes do not know is which ones are currently up and which ones are currently down. The goal of an election algorithm is to ensure that when an election starts, it concludes with all processes agreeing on who the new coordinator is to be.

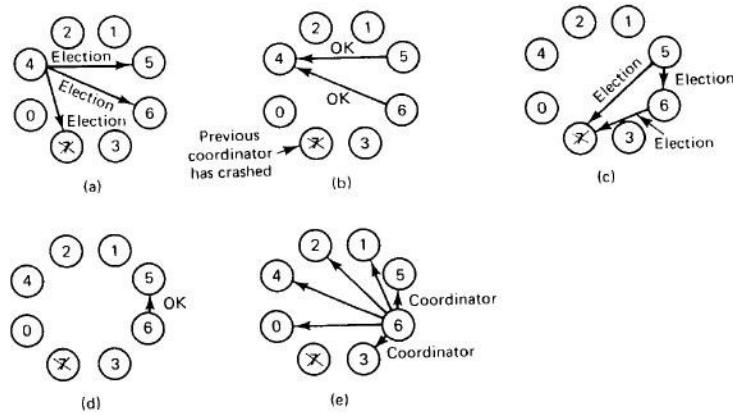
#### **1.3.3.1 The Bully Algorithm**

As a first example, consider the bully algorithm devised by Garcia-Molina. When a process notices that the coordinator is no longer responding to requests, it initiates an election. A process, P, holds an election as follows:

1. P sends an ELECTION message to all processes with higher numbers.
2. If no one responds, P wins the election and becomes coordinator.
3. If one of the higher-ups answers, it takes over. P's job is done.

At any moment, a process can get an ELECTION message from one of its lower-numbered colleagues. When such a message arrives, the receiver sends an OK message back to the sender to indicate that he is alive and will take over. The receiver then holds an election, unless it is already holding one. Eventually, all processes give up but one, and that one is the new coordinator. It announces its victory by sending all processes a message telling them that starting immediately it is the new coordinator. If a process that was previously down comes back up, it holds an election. If it happens to be the highest-numbered process currently running, it will win the election and take over the coordinator's job. Thus the biggest guy in town always wins, hence the name "bully algorithm."

In Fig. 1.17 we see an example of how the bully algorithm works. The group consists of eight processes, numbered from 0 to 7. Previously process 7 was the coordinator, but it has just crashed. Process 4 is the first one to notice this, so it sends ELECTION messages to all the processes higher than it, namely Processes 5 and 6 both respond with OK, as shown in Fig. 1.17(a). Upon getting the first of these responses, 4 knows that its job is over. It knows that one of these bigwigs will take over and become coordinator. It just sits back and waits to see who the winner will be (although at this point it can make a pretty good guess).



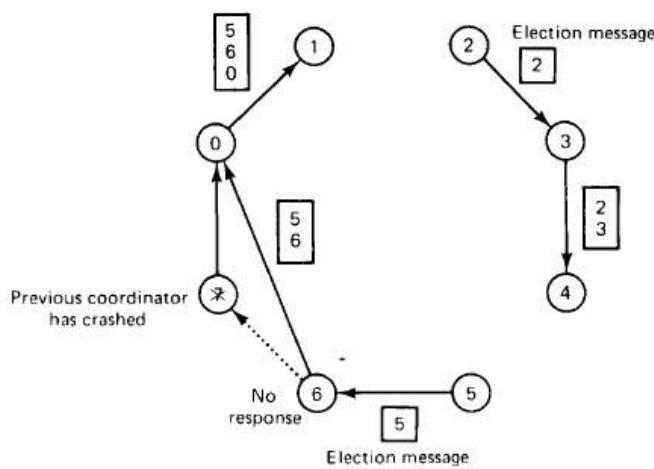
**Fig. 1.17. The bully election algorithm.** (a) Process 4 holds an election. (b) Processes 5 and 6 respond, telling 4 to stop. (c) Now 5 and 6 each hold an election. (d) Process 6 tells 5 to stop. (e) Process 6 wins & tells everyone.

In Fig. 1.17(c) both 5 and 6 hold elections, each one only sending messages to those processes higher than itself. In Fig. 1.17(d) process 6 tells 5 that it will take over. At this point 6 knows that 7 is dead and that it (6) is the winner. If there is state information to be collected from disk or elsewhere to pick up where the old coordinator left off, 6 must now do what is needed. When it is ready to take over, 6 announces this by sending a COORDINATOR message to all running processes. When 4 gets this message, it can now continue with the operation it was trying to do when it discovered that 7 was dead, but using 6 as the coordinator this time. In this way the failure of 7 is handled and the work can continue. If process 7 is ever restarted, it will just send all the others a COORDINATOR message and bully them into submission.

### 1.3.3.2 A Ring Algorithm

Another election algorithm is based on the use of a ring, but without a token. We assume that the processes are physically or logically ordered, so that each process knows who its successor is. When any process notices that the coordinator is not functioning, it builds an ELECTION message containing its own process number and sends the message to its successor. If the successor is down, the sender skips over the successor and goes to the next member along the ring, or the one after that, until a running process is located. At each step, the sender adds its own process number to the list in the message.

Eventually, the message gets back to the process that started it all. That process recognizes this event when it receives an incoming message containing its own process number. At that point, the message type is changed to COORDINATOR and circulated once again, this time to inform everyone else who the coordinator is (the list member with the highest number) and who the members of the new ring are. When this message has circulated once, it is removed and everyone goes back to work.



**Fig. 1.18. Election algorithm using a ring.**

In Fig. 1.18 we see what happens if two processes, 2 and 5, discover simultaneously that the previous coordinator, process 7, has crashed. Each of these builds an ELECTION message and starts circulating it. Eventually, both messages will go all the way around, and both 2 and 5 will convert them into COORDINATOR messages, with exactly the same members and in the same order. When both have gone around again, both will be removed. It does no harm to have extra messages circulating; at most it wastes a little bandwidth.

### 1.3.4 Atomic Transactions

All the synchronization techniques we have studied so far are essentially low level, like semaphores. They require the programmer to be intimately involved with all the details of mutual exclusion, critical region management, deadlock prevention, and crash recovery. What we would really like is a much higher-level abstraction, one that hides these technical issues and allows the programmer to concentrate on the algorithms and how the processes work together in parallel. Such an abstraction exists and is widely used in distributed systems. We will call it an atomic transaction, or simply transaction. The term atomic action is also widely used.

One process announces that it wants to begin a transaction with one or more other processes. They can negotiate various options, create and delete objects, and operations for a while. Then the initiator announces that it wants all the others to commit themselves to the work done so far. If all of them agree, the results are made permanent. If one or more processes refuse (or crash before agreement), the situation reverts to exactly the state it had before the transaction began, with all side effects on objects, files, data bases, and so on, magically wiped out. This all-or-nothing property eases the programmer's job.

Now look at a modern banking application that updates an online data base in place. The customer calls up the bank using a PC with the intention of withdrawing money from one account and depositing it in another. The operation is performed in two steps:

1. withdraw(amount, account1)
2. deposit(amount, account2).

If the connection is broken after the first one but before the second one, the first account will have been debited but the second one will not have been credited. The money vanishes into thin air.

Being able to group these two operations in an atomic transaction would solve the problem. Either both would be completed, or neither would be completed. The key is rolling back to the initial state if the transaction fails to complete. What we really want is a way to rewind the data base as we could the magnetic tapes. This ability is what the atomic transaction has to offer.

#### 1.3.4.1 The Transaction Model

We will now develop a more precise model of what a transaction is and what its properties are. The system is assumed to consist of some number of independent processes, each of which can fail at random. Communication is normally unreliable in that messages can be lost, but lower levels can use a timeout and retransmission protocol to recover from lost messages. Thus for this discussion we can assume that communication errors are handled transparently by underlying software.

Storage comes in three categories. First we have ordinary RAM memory, which is wiped out when the power fails or a machine crashes. Next we have disk storage, which survives CPU failures but which can be lost in disk head crashes. Finally, we have *stable storage*, which is designed to survive anything except major calamities such as floods and earthquakes. Stable storage can be implemented with a pair of ordinary disks, as shown in Fig. 1.19(a). Each block on drive 2 is an exact copy of the corresponding block on drive 1. When a block is updated, first the block on drive 1 is updated and verified, then the same block on drive 2 is done.

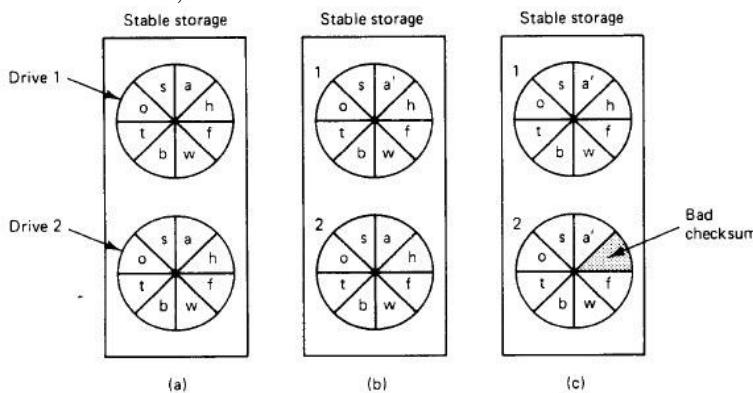


Fig. 1.19. (a) Stable storage. (b) Crash after drive 1 is updated. (c) Bad spot.

Suppose that the system crashes after drive 1 is updated but before drive 2 is updated, as shown in Fig. 1.19(b). Upon recovery, the disk can be compared block for block. Whenever two corresponding blocks differ, it can be assumed that drive 1 is the correct one (because drive 1 is always updated before drive 2), so the new block is copied from drive 1 to drive 2. When the recovery process is complete, both drives will again be identical.

Another potential problem is the spontaneous decay of a block. Dust particles or general wear and tear can give a previously valid block a sudden checksum error, without cause or warning, as shown in Fig. DOS.F11(c). When such an error is detected, the bad block can be regenerated from the corresponding block on the other drive.

As a consequence of its implementation, stable storage is well suited to applications that require a high degree of fault tolerance, such as atomic transactions. When data are written to stable storage and then read back to check that they have been written correctly, the chance of them subsequently being lost is extremely small.

### 1.3.4.2 Transaction Primitives

Programming using transactions requires special primitives that must either be supplied by the operating system or by the language runtime system. Examples are:

1. BEGIN-TRANSACTION: Mark the start of a transaction.
2. END-TRANSACTION: Terminate the transaction and try to commit.
3. ABORT-TRANSACTION: Kill the transaction; restore the old values.
4. READ: Read data from a file (or other object).
5. WRITE: Write data to a file (or other object).

The exact list of primitives depends on what kinds of objects are being used in the transaction. In a mail system, there might be primitives to send, receive, and forward mail. In an accounting system, they might be quite different. READ and WRITE are typical examples, however. Ordinary statements, procedure calls, and so on, are also allowed inside a transaction.

BEGIN-TRANSACTION and END-TRANSACTION are used to delimit the scope of a transaction. The operations between them form the body of the transaction. Either all of them are executed or none are executed. These may be system calls, library procedures, or bracketing statements in a language, depending on the implementation.

Consider, for example, the process of reserving a seat from Timisoara, to Lyon, in an airline reservation system. One route is Timisoara to Bucuresti, Bucuresti to Paris, and Paris to Lyon. In what follows we see reservations for these three separate flights being made as three actions.

```
BEGIN-TRANSACTION
    reserve TSR - OTP;
    reserve OTP - CDG;
    reserve CDG - LYN;
END-TRANSACTION
```

```
BEGIN-TRANSACTION
    reserve TSR - OTP;
    reserve OTP - CDG;
    CDG - LYN full ->ABORT-TRANSACTION;
```

Now suppose that the first two flights have been reserved but the third one is booked solid. The transaction is aborted and the results of the first two bookings are undone—the airline data base is restored to the value it had before the transaction started. It is as though nothing happened.

**Properties of Transactions.** Transactions have four essential properties. Transactions are:

1. Atomic: To the outside world, the transaction happens indivisibly.
2. Consistent: The transaction does not violate system invariants.
3. Isolated: Concurrent transactions do not interfere with each other.
4. Durable: Once a transaction commits, the changes are permanent.

These properties are often referred to by their initial letters, ACID.

The first key property exhibited by all transactions is that they are *atomic*. This property ensures that each transaction either happens completely, or not at all, and if it happens, it happens in a single indivisible, instantaneous action. While a transaction is in progress, other processes (whether or not they are themselves involved in transactions) cannot see any of the intermediate states.

Suppose, for example, that some file is 10 bytes long when a transaction starts to append to it. If other processes read the file while the transaction is in progress, they see only the original 10 bytes, no matter how many bytes the transaction has appended. If the transaction commits successfully, the file grows instantaneously to its new size at the moment of commitment, with no intermediate states, no matter how many operations it took to get it there.

The second property says that they are *consistent*. What this means is that if the system has certain invariants that must always hold, if they held before the transaction, they will hold afterward too. For example, in a banking

system, a key invariant is the law of conservation of money. After any internal transfer, the amount of money in the bank must be the same as it was before the transfer, but for a brief moment during the transaction, this invariant may be violated. The violation is not visible outside the transaction, however.

The third property says that transactions are *isolated or serializable*. What it means is that if two or more transactions are running at the same time, to each of them and to other processes, the final result looks as though all transactions ran sequentially in some (system dependent) order.

Suppose we have three transactions that are executed simultaneously by three separate processes. If they were to be run sequentially, the final value of  $x$  would be 1, 2, or 3, depending which one ran last ( $x$  could be a shared variable, a file, or some other kind of object).

BEGIN-TRANSACTION	BEGIN-TRANSACTION	BEGIN-TRANSACTION
$x = 0;$	$x = 0;$	$x = 0;$
$x = x+1;$	$x = x+2;$	$x = x+3;$
END-TRANSACTION	END-TRANSACTION	END-TRANSACTION

Consider the following various orders, called *schedules*, in which they might be interleaved.

	Time ->	
Schedule 1	$x=0; x=x+1; x=0; x=x+2; x=0; x=x + 3;$	Legal
Schedule 2	$x=0; x=0; x=x+1; x=x+2; x=0; x=x + 3;$	Legal
Schedule 3	$x=0; x=0; x:=x+1; x=0; x=x + 2; x=x + 3;$	Illegal

Schedule 1 is actually serialized. In other words, the transactions run strictly sequentially, so it meets the serializability condition by definition. Schedule 2 is not serialized, but is still legal because it results in a value for  $x$  that could have been achieved by running the transactions strictly sequentially. The third one is illegal since it sets  $x$  to 5, something that no sequential order of the transactions could produce. It is up to the system to ensure that individual operations are interleaved correctly. By allowing the system the freedom to choose any ordering of the operations it wants to--provided that it gets the answer right--we eliminate the need for programmers to do their own mutual exclusion, thus simplifying the programming.

The fourth property says that transactions are *durable*. It refers to the fact that once a transaction commits, no matter what happens, the transaction goes forward and the results become permanent. No failure after the commit can undo the results or cause them to be lost.

**Nested Transactions.** Transactions may contain subtransactions, often called nested transactions. The top-level transaction may fork off children that run in parallel with one another, on different processors, to gain performance or simplify programming. Each of these children may execute one or more subtransactions, or fork off its own children.

Subtransactions give rise to a subtle, but important, problem. Imagine that a transaction starts several subtransactions in parallel, and one of these commits, making its results visible to the parent transaction. After further computation, the parent aborts, restoring the entire system to the state it had before the level transaction started. Consequently, the results of the subtransaction that committed must nevertheless be undone. Thus the permanence referred to above applies only to top-level transactions.

Since transactions can be nested arbitrarily deeply, considerable administration is needed to get everything right. The semantics are clear, however. When any transaction or subtransaction starts, it is conceptually given a private copy of all objects in the entire system for it to manipulate as it wishes. If it aborts, its private universe just vanishes, as if it had never existed. If it commits, its private universe replaces the parent's universe. Thus if a subtransaction commits and then later a new subtransaction is started, the second one sees the results produced by the first one.

### 1.3.4.3 Implementation

Transactions sound like a great idea, but how are they implemented? That is the question we will tackle in this section. It should be clear by now that if each process executing a transaction just updates the objects it uses (files, data base records, etc.) in place, transactions will not be atomic and changes will not vanish magically if the

transaction aborts. Furthermore, the results of running multiple transactions will not be serializable either. Clearly, some other implementation method is required. Two methods are commonly used.

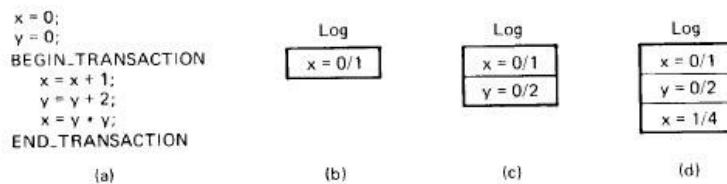
**Private Workspace.** Conceptually, when a process starts a transaction, it is given a private workspace containing all the files (and other objects) to which it has access. Until the transaction either commits or aborts, all of its reads and writes go to the private workspace, rather than the "real" one, by which we mean the normal file system. This observation leads directly to the first implementation method: actually giving a process a private workspace at the instant it begins a transaction.

The problem with this technique is that the cost of copying everything to a private workspace is prohibitive, but various optimizations make it feasible. The first optimization is based on the realization that when a process reads a file but does not modify it, there is no need for a private copy. It can just use the real one (unless it has been changed since the transaction started). Consequently, when a process starts a transaction, it is sufficient to create a private workspace for it that is empty except for a pointer back to its parent's workspace. When the transaction is at the top level, the parent's workspace is the "real" file system. When the process opens a file for reading, the back pointers are followed until the file is located in the parent's (or further ancestor's) workspace.

When a file is opened for writing, it can be located in the same way as for reading, except that now it is first copied to the private workspace. However, a second optimization removes most of the copying, even here. Instead of copying the entire file, only the file's index is copied into the private workspace. The index is the block of data associated with each file telling where its disk blocks are. In Unix the index is the i-node. Using the private index, the file can be read in the usual way, since the disk addresses it contains are for the original disk blocks. However, when a file block is first modified, a copy of the block is made and the address of the copy inserted into the index. The block can then be updated without affecting the original. Appended blocks are handled this way too. The new blocks are sometimes called *shadow blocks*.

The process running the transaction sees the modified file, but all other processes continue to see the original file. In a more complex transaction, the private workspace might contain a large number of files instead of just one. If the transaction aborts, the private workspace is simply deleted and all the private blocks that it points to are put back on the free list. If the transaction commits, the private indices are moved into the parent's workspace atomically. The blocks that are no longer reachable are put onto the free list.

**Writeahead Log.** The other common method of implementing transactions is the writeahead log, sometimes called an intentions list. With this method, files are actually modified in place, but before any block is changed, a record is written to the writeahead log on stable storage telling which transaction is making the change, which file and block is being changed, and what the old and new values are. Only after the log has been written successfully is the change made to the file.



**Fig. 1.20. (a) A transaction. The log before each statement is executed.**

Figure 1.20 gives an example of how the log works. In Fig. 1.20(a) we have a simple transaction that uses two shared variables (or other objects), x and y, both initialized to 0. For each of the three statements inside the transaction, a record is written before executing the statement, giving the old and new values, separated by a slash.

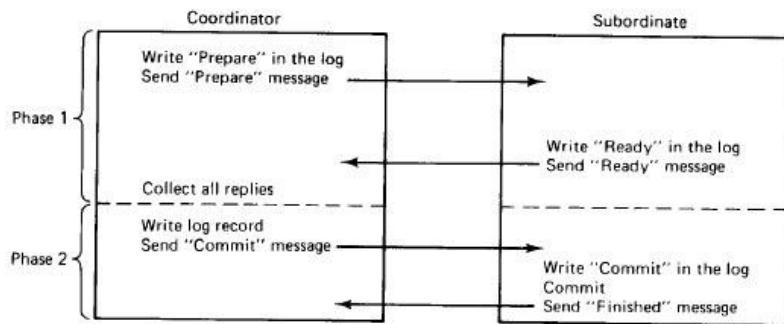
If the transaction succeeds and is committed, a commit record is written to the log, but the data structures do not have to be changed, as they have already been updated. If the transaction aborts, the log can be used to back up to the original state. Starting at the end and going backward, each log record is read and the change described in it undone. This action is called a *rollback*.

The log can also be used for recovering from crashes. Suppose that the process doing the transaction crashes just after having written the last log record of Fig. 1.20(d), but before changing x. After the failed machine is rebooted, the log is checked to see if any transactions were in progress at the time of the crash. When the last record is read and the current value of x is seen to be 1, it is clear that the crash occurred before the update was made, so x is set

to 4. If, on the other hand,  $x$  is 4 at the time of recovery, it is equally clear that the crash occurred after the update, so nothing need be changed. Using the log, it is possible to go forward (do the transaction) or go backward (undo the transaction).

**Two-Phase Commit Protocol.** As we have pointed out repeatedly, the action of committing a transaction must be done atomically, that is, instantaneously and indivisibly. In a distributed system, the commit may require the cooperation of multiple processes on different machines, each of which holds some of the variables, files, and data bases, and other objects changed by the transaction. In this section we will study a protocol for achieving atomic commit in a distributed system.

The protocol we will look at is called the *two-phase commit protocol*. Although it is not the only such protocol, it is probably the most widely used. The basic idea is illustrated in Fig. 1.21. One of the processes involved functions as the coordinator. Usually, this is the one executing the transaction. The commit protocol begins when the coordinator writes a log entry saying that it is starting the commit protocol, followed by sending each of the other processes involved (the subordinates) a message telling them to prepare to commit.



*Fig. 1.21. The two-phase commit protocol when it succeeds.*

When a subordinate gets the message it checks to see if it is ready to commit, makes a log entry, and sends back its decision. When the coordinator has received all the responses, it knows whether to commit or abort. If all the processes are prepared to commit, the transaction is committed. If one or more are unable to commit (or do not respond), the transaction is aborted. Either way, the coordinator writes a log entry and then sends a message to each subordinate informing it of the decision. It is this write to the log that actually commits the transaction and makes it go forward no matter what happens afterward.

Due to the use of the log on stable storage, this protocol is highly resilient in the face of (multiple) crashes. If the coordinator crashes after having written the initial log record, upon recovery it can just continue where it left off, repeating the initial message if need be. If it crashes after having written the result of the vote to the log, upon recovery it can just tell all the subordinates of the result. If a subordinate crashes before having replied to the first message, the coordinator will keep sending it messages, until it gives up. If it crashes later, it can see from the log where it was, and thus what it must do.

#### 1.3.4.4 Concurrency Control

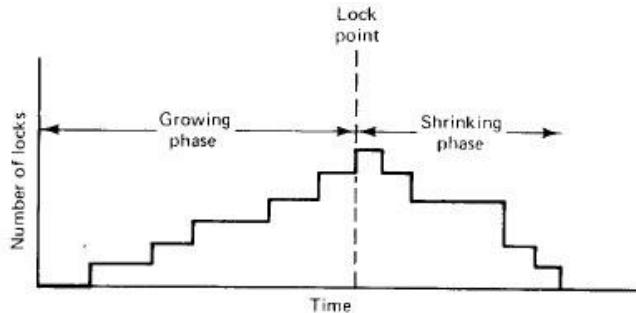
When multiple transactions are executing simultaneously in different processes (on different processors), some mechanism is needed to keep them out of each other's way. That mechanism is called a concurrency control algorithm. In this section we will study three different ones.

**Locking.** The oldest and most widely used concurrency control algorithm is locking. In the simplest form, when a process needs to read or write a file (or other object) as part of a transaction, it first locks the file. Locking can be done using a single centralized lock manager, or with a local lock manager on each machine for managing local files. In both cases the lock manager maintains a list of locked files, and rejects all attempts to lock files that are already locked by another process. Since well-behaved processes do not attempt to access a file before it has been locked, setting a lock on a file keeps everyone else away from it and thus ensures that it will not change during the lifetime of the transaction. Locks are normally acquired and released by the transaction system and do not require action by the programmer.

This basic scheme is overly restrictive and can be improved by distinguishing read locks from write locks. If a read lock is set on a file, other read locks are permitted. Read locks are set to make sure that the file does not change (excluding all writers), but there is no reason to forbid other transactions from reading the file. In contrast,

when a file is locked for writing, no other locks of any kind are permitted. Thus read locks are shared, but write locks must be exclusive.

For simplicity, we have assumed that the unit of locking is the entire file. In practice, it might be a smaller item, such as an individual record or page, or a larger item, such as an entire data base. The issue of how large an item to lock is called the *granularity of locking*. The finer the granularity, the more precise the lock can be, and the more parallelism can be achieved (e.g. by not blocking a process that wants to use the end of a file just because some other process is using the beginning). On the other hand, fine-grained locking requires more locks, is more expensive, and is more likely to lead to deadlocks.



*Fig. 1.22. Two-phase locking.*

Acquiring and releasing locks precisely at the moment they are needed or no longer needed can lead to inconsistency and deadlocks. Instead, most transactions that are implemented by locking use what is called two-phase locking. In two-phase locking, which is illustrated in Fig. DOS.F14, the process first acquires all the locks it needs during the growing phase, then releases them during the shrinking phase. If the process refrains from updating any files until it reaches the shrinking phase, failure to acquire some lock can be dealt with simply by releasing all locks, waiting a little while, and starting all over. Furthermore, it can be proven that if all transactions use two-phase locking, all schedules formed by interleaving them are serializable. This is why two-phase locking is widely used.

In many systems, the shrinking phase does not take place until the transaction has finished running and has either committed or aborted. This policy, called strict two-phase locking, has two main advantages. First, a transaction always reads a value written by a committed transaction; therefore, one never has to abort a transaction because its calculations were based on a file it should not have seen. Second, all lock acquisitions and releases can be handled by the system without the transaction being aware of them: locks are acquired whenever a file is to be accessed and released when the transaction has finished. This policy eliminates cascaded aborts: having to undo a committed transaction because it saw a file it should not have seen.

Locking, even two-phase locking, can lead to deadlocks. If two processes each try to acquire the same pair of locks but in the opposite order, a deadlock may result. The usual techniques apply here, such as acquiring all locks in some canonical order to prevent hold-and-wait cycles. Also possible is deadlock detection by maintaining an explicit graph of which process has which locks and wants which locks, and checking the graph for cycles. Finally, when it is known in advance that a lock will never be held longer than T sec, a timeout scheme can be used: if a lock remains continuously under the same ownership for longer than T sec, there must be a deadlock.

**Optimistic Concurrency Control.** A second approach to handling multiple transactions at the same time is *optimistic concurrency control*. The idea behind this technique is surprisingly simple: just go ahead and do whatever you want to, without paying attention to what anybody else is doing. If there is a problem, about it later. In practice, conflicts are relatively rare, so most of the time it works all right.

Although conflicts may be rare, they are not impossible, so some way is needed to handle them. What optimistic concurrency control does is keep track of which files have been read and written. At the point of committing, it checks all other transactions to see if any of its files have been changed since the transaction started. If so, the transaction is aborted. If not, it is committed.

Optimistic concurrency control fits best with the implementation based on private workspaces. That way, each transaction changes its files privately, without interference from the others. At the end, the new files are either committed or released.

The big advantages of optimistic concurrency control are that it is deadlock free and allows maximum parallelism because no process ever has to wait for a lock. The disadvantage is that sometimes it may fail, in which case the transaction has to be run all over again. Under conditions of heavy load, the probability of failure may go up substantially, making optimistic concurrency control a poor choice.

**Timestamps.** A completely different approach to concurrency control is to assign each transaction a timestamp at the moment it does BEGIN-TRANSACTION. Using Lamport's algorithm, we can ensure that the timestamps are unique, which is important here. Every file in the system has a read timestamp and a write timestamp associated with it, telling which committed transaction last read and wrote it, respectively. If transactions are short and widely spaced in time, it will normally occur that when a process tries to access a file, the file's read and write timestamps will be lower (older) than the current transaction's timestamp. This ordering means that the transactions are being processed in the proper order, so everything is all right.

When the ordering is incorrect, it means that a transaction that started later than the current one has managed to get in there, access the file, and commit. This situation means that the current transaction is too late, so it is aborted. In a sense, this mechanism is also optimistic, like that of Kung and Robinson, although the details are quite different. In Kung and Robinson's method, we are hoping that concurrent transactions do not use the same files. In the timestamp method, we do not mind if concurrent transactions use the same files, as long as the lower numbered transaction always goes first.

It is easiest to explain the timestamp method by means of an example. Imagine that there are three transactions, *a*, *b*, and *g*. *a* ran a long time ago, and used every file needed by *b* and *g*, so all their files have read and write timestamps set to *a*'s timestamp. *b* and *g* start concurrently, with *b* having a lower timestamp than *g* (but higher than *a*, of course).

Let us first consider beta writing a file. Call its timestamp, *T*, and the read and write timestamps of the file to be written *T<sub>RD</sub>* and *T<sub>WR</sub>*, respectively. Unless *g* has snuck in already and committed, both *T<sub>RD</sub>* and *T<sub>WR</sub>* will be alpha's timestamp, and thus less than *T*. In Fig. 1.23(a) and (b) we see that *T* is larger than both *T<sub>RD</sub>* and *T<sub>WR</sub>* (*g* has not already committed), so the write is accepted and done tentatively. It will become permanent when *b* commits. *b*'s timestamp is now recorded in the file as a tentative write.

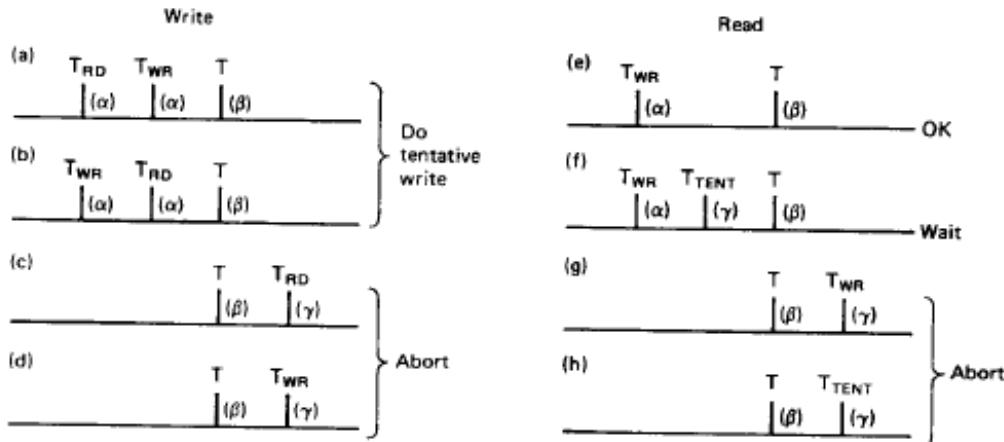


Fig. 1.23. Concurrency control using timestamps.

In Fig. 1.23(c) and (d) *b* is out of luck. Gamma has either read (c) or written (d) the file and committed. *b*'s transaction is aborted. However, it can apply for a new timestamp and start all over again. Now look at reads. In Fig. 1.23(e), there is no conflict, so the read can happen immediately. In Fig. 1.23(f), some interloper has gotten in there and is trying to write the file. The interloper's timestamp is lower than *b*'s, so *b* simply waits until the interloper commits, at which time it can read the new file and continue. In Fig. 1.23(g), *g* has changed the file and already committed. Again *b* must abort. In Fig. 1.23(h), *g* is in the process of changing the file, although it has not committed yet. Still, *b* is too late and must abort.

Timestamping has different properties than locking. When a transaction encounters a larger (later) timestamp, it aborts, whereas under the same circumstances with locking it would either wait or be able to proceed immediately. On the other hand, it is deadlock free, which is a big plus.

All in all, transactions offer many advantages and thus are a promising technique for building reliable distributed systems. Their chief problem is their great implementation complexity, which yields low performance. These problems are being worked on, and perhaps in due course they will be solved.

### 1.3.5 Deadlocks in Distributed Systems

Deadlocks in distributed systems are similar to deadlocks in processor systems, only worse. They are harder to avoid, prevent, or even detect, and harder to cure when tracked down because all the relevant information is scattered over many machines. In some systems, such as distributed data base systems, they can be extremely serious, so it is important to understand how they differ from ordinary deadlocks and what can be done about them.

Some people make a distinction between two kinds of distributed deadlocks: *communication deadlocks* and *resource deadlocks*. A communication deadlock occurs, for example, when process A is trying to send a message to process B, which in turn is trying to send one to process C, which is trying to send one to A. There are various scenarios in which this situation leads to deadlock, such as no buffers being available. A resource deadlock occurs when processes are fighting over exclusive access to I/O devices, files, locks, or other resources.

We will not make that distinction here, since communication channels, buffers, and so on, are also resources and can be modeled as resource deadlocks because processes can request them and release them. Furthermore, circular communication patterns of the type just described are quite rare in most systems. In client-server systems, for example, a client might send a message (or perform an RPC) with a file server, which might send a message to a disk server. However, it is unlikely that the disk server, acting as a client, would send a message to the original client, expecting it to act like a server. Thus the circular wait condition is unlikely to occur as a result of communication alone.

Various strategies are used to handle deadlocks. Four of the best-known ones are listed and discussed below.

1. The ostrich algorithm (ignore the problem).
2. Detection (let deadlocks occur, detect them, and try to recover).
3. Prevention (statically make deadlocks structurally impossible).
4. Avoidance (avoid deadlocks by allocating resources carefully).

All four are potentially applicable to distributed systems. The ostrich algorithm is as good and as popular in distributed systems as it is in single-processor systems. In distributed systems used for programming, office automation, process control, and many other applications, no system-wide deadlock mechanism is present, although individual applications, such as distributed data bases, can implement their own if they need one.

Deadlock detection and recovery is also popular, primarily because prevention and avoidance are so difficult. We will discuss several algorithms for deadlock detection below. Deadlock prevention is also possible, although more difficult than in uni-processor systems. However, in the presence of atomic transactions, some new options become available. Two algorithms are discussed below.

Finally, deadlock avoidance is never used in distributed systems. It is not even used in single-processor systems, so why should it be used in the more difficult case of distributed systems? The problem is that the banker's algorithm and similar algorithms need to know (in advance) how much of each resource every process will eventually need. This information is rarely, if ever, available. Thus our discussion of deadlocks in distributed systems will focus on just two of the techniques: deadlock detection and deadlock prevention.

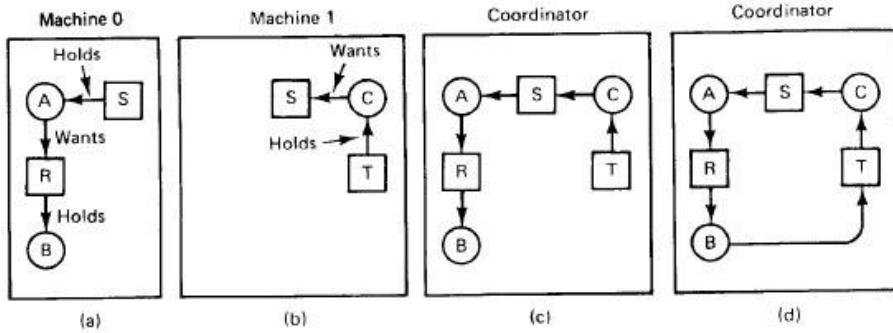
**Distributed Deadlock Detection.** Finding general methods for preventing or avoiding distributed deadlocks appears to be quite difficult, so many researchers have tried to deal with the simpler problem of just detecting deadlocks, rather than trying to inhibit their occurrence.

However, the presence of atomic transactions in some distributed systems makes a major conceptual difference. When a deadlock is detected in a conventional operating system, the way to resolve it is to kill off one or more processes. Doing so invariably leads to one or more unhappy users. When a deadlock is detected in a system based on atomic transactions, it is resolved by aborting one or more transactions. But as we have seen in detail above, transactions have been designed to withstand being aborted. When a transaction is aborted because it contributes to a deadlock, the system is first restored to the state it had before the transaction began, at which point the transaction can start again. With a little bit of luck, it will succeed the second time. Thus the difference is that the

consequences of killing off a process are much less severe when transactions are used than when they are not used.

**Centralized Deadlock Detection.** As a first attempt, we can use a centralized deadlock detection algorithm and try to imitate the non-distributed algorithm. Although each machine maintains the resource graph for its own processes and resources, a central coordinator maintains the resource graph for the entire system (the union of all the individual graphs). When the coordinator detects a cycle, it kills off one process to break the deadlock.

Unlike the centralized case, where all the information is automatically available in the right place, in a distributed system it has to be sent there explicitly. Each machine maintains the graph for its own processes and resources. Several possibilities exist for getting it there. First, whenever an arc is added or deleted from the resource graph, a message can be sent to the coordinator providing the update. Second, periodically, every process can send a list of arcs added or deleted since the previous update. This method requires fewer messages than the first one. Third, the coordinator can ask for information when it needs it.



*Fig. 1.24. (a) Initial resource graph for machine 0. (b) Initial resource graph for machine 1.(c) The coordinator's view of the world. (d) The situation after the delayed message.*

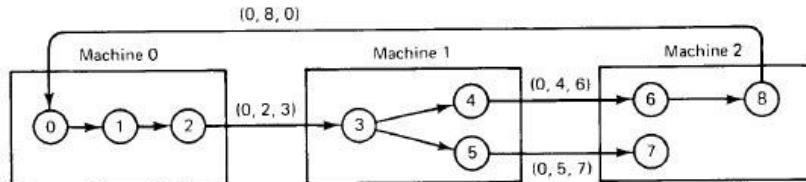
Unfortunately, none of these methods work well. Consider a system with processes A and B running on machine 0, and process C running on machine 1. Three resources exist: R, S and T. Initially, the situation is as shown in Fig. 1.24(a) and (b): A holds S but wants R, which it cannot have because B is using it; C has T and wants S, too. The coordinator's view of the world is shown in Fig. 1.24(c). This configuration is safe. As soon as B finishes, A can get R and finish, releasing S for C.

After a while, B releases R and asks for T, a perfectly legal and safe swap. Machine 0 sends a message to the coordinator announcing the release of R, and machine 1 sends a message to the coordinator announcing the fact that B is now waiting for its resource, T. Unfortunately, the message from machine 1 arrives first, leading the coordinator to construct the graph of Fig. 1.24(d). The coordinator incorrectly concludes that a deadlock exists and kills some process. Such a situation is called a false deadlock. Many deadlock algorithms in distributed systems produce false deadlocks like this due to incomplete or delayed information.

One possible way out might be to use Lamport's algorithm to provide global time. Since the message from machine 1 to the coordinator is triggered by the request from machine 0, the message from machine 1 to the coordinator will indeed have a later timestamp than the message from machine 0 to the coordinator. When the coordinator gets the message from machine 1 that leads it to suspect deadlock, it could send a message to every machine in the system saying: "I just received a message with timestamp T which leads to deadlock. If anyone has a message for me with an earlier timestamp, please send it immediately." When every machine has replied, positively or negatively, the coordinator will see that the arc from R to B has vanished, so the system is still safe. Although this method eliminates the false deadlock, it requires global time and is expensive. Furthermore, other situations exist where eliminating false deadlock is much harder.

**Distributed Deadlock Detection.** Many distributed deadlock detection algorithms have been published. In the algorithm of Chandy-Misra-Haas, processes are allowed to request multiple resources (e.g. locks) at once, instead of one at a time. By allowing multiple requests simultaneously, the growing phase of a transaction can be speeded up considerably. The consequence of this change to the model is that a process may now wait on two or more resources simultaneously.

In Fig. 1.25, we present a modified resource graph, where only the processes are shown. Each arc passes through a resource, as usual, but for simplicity the resources have been omitted from the figure. Notice that process 3 on machine 1 is waiting for two resources, one held by process 4 and one held by process 5.



**Fig. 1.25. The Chandy-Misra-Haas distributed deadlock detection algorithm.**

Some of the processes are waiting for local resources, such as process 1, but others, such as process 2, are waiting for resources that are located on a different machine. It is precisely these cross-machine arcs that make looking for cycles difficult. The Chandy-Misra-Haas algorithm is invoked when a process has to wait for some resource, for example, process 0 blocking on process 1. At that point a special probe message is generated and sent to the process (or processes) holding the needed resources. The message consists of three numbers: the process that just blocked, the process sending the message, and the process to whom it is being sent. The initial message from 1 to 0 contains the triple (0, 0, 1).

When the message arrives, the recipient checks to see if it itself is waiting for any processes. If so, the message is updated, keeping the first field but replacing the second field by its own process number and the third one by the number of the process it is waiting for. The message is then sent to the process on which it is blocked. If it is blocked on multiple processes, all of them are sent (different) messages. This algorithm is followed whether the resource is local or remote. In Fig. 1.25 we see the remote messages labeled (0, 2, 3), (0, 4, 6), (0, 5, 7), and (0, 8, 0). If a message goes all the way around and comes back to the original sender, that is, the process listed in the first field, a cycle exists and the system is deadlocked.

There are various ways in which the deadlock can be broken. One way is to have the process that initiated the probe commit suicide. However, this method has problems if several processes invoke the algorithm simultaneously. In Fig. 1.25, for example, imagine that both 0 and 6 block at the same moment, and both initiate probes. Each would eventually discover the deadlock, and each would kill itself. This is overkill. Getting rid of one of them is enough. An alternative algorithm is to have each process add its identity to the end of the probe message so that when it returned to the initial sender, the complete cycle would be listed. The sender can then see which process has the highest number, and kill that one or send it a message asking it to kill itself. Either way, if multiple processes discover the same cycle at the same time, they will all choose the same victim.

There are few areas of computer science in which theory and practice diverge as much as in distributed deadlock detection algorithms. Discovering yet another deadlock detection algorithm is the goal of many researchers. Some of the proposed algorithms require processes to send probes when they are blocked. However, sending a probe when you are blocked is not entirely trivial.

**Distributed Deadlock Prevention.** Deadlock prevention consists of carefully designing the system so that deadlocks are structurally impossible. Various techniques include allowing processes to hold only one resource at a time, requiring processes to request all their resources initially, and making processes release all resources when asking for a new one. All of these are cumbersome in practice. A method that sometimes works is to order all the resources and require processes to acquire them in strictly increasing order. This approach means that a process can never hold a high resource and ask for a low one, thus making cycles impossible.

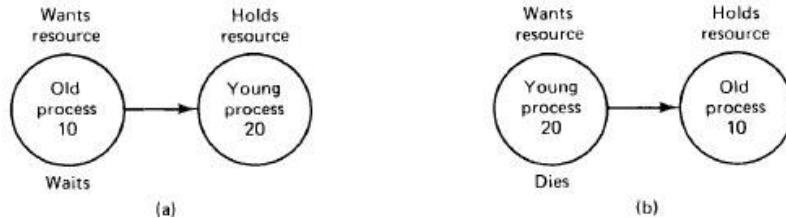
However, in a distributed system with global time and atomic transactions, two other practical algorithms are possible. Both are based on the idea of assigning each transaction a global timestamp at the moment it starts. As in many timestamp-based algorithms, in these two it is essential that no two transactions are ever assigned exactly the same timestamp. As we have seen, Lamport's algorithm guarantees uniqueness (effectively by using process numbers to break ties).

The idea behind the algorithm is that when one process is about to block waiting for a resource that another process is using, a check is made to see which has a larger timestamp (i.e. is younger). We can then allow the wait only if the waiting process has a lower timestamp (is older) than the process waited for. In this manner, following any chain of waiting processes, the timestamps always increase, so cycles are impossible. Alternatively, we can allow processes to wait only if the waiting process has a higher timestamp (is younger) than the process waited for, in which case the timestamps decrease along the chain.

Although both methods prevent deadlocks, it is wiser to give priority to older processes. They have run longer, so the system has a larger investment in them, and they are likely to hold more resources. Also, a young process that

is killed off will eventually age until it is the oldest one in the system, so this choice eliminates starvation. Killing a transaction is relatively harmless, since by definition it can be restarted safely later.

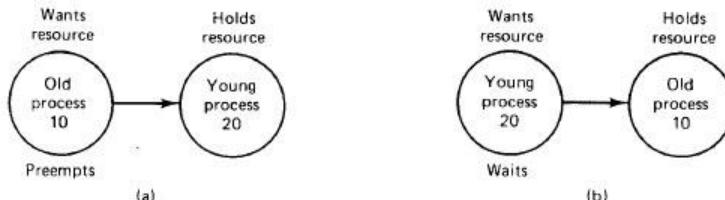
To make this algorithm clearer, consider the situation of Fig. 1.26. In (a), an old process wants a resource held by a young process. In (b), a young process wants a resource held by an old process. In one case we should allow the process to wait; in the other we should kill it. Suppose that we label (a) dies and (b) wait. Then we are killing off an old process trying to use a resource held by a young process, which is inefficient. Thus we must label it the other way, as shown in the figure. Under these conditions, the arrows always point in the direction of increasing transaction numbers, making cycles impossible. This algorithm is called *wait-die*.



**Fig. 1.26. The wait-die deadlock prevention algorithm.**

Once we are assuming the existence of transactions, we can do something that had previously been forbidden: take resources away from running processes. In effect we are saying that when a conflict arises, instead of killing the process making the request, we can kill the resource owner. Without transactions, killing a process might have severe consequences, since the process might have modified files, for example. With transactions, these effects will vanish magically when the transaction dies.

Now consider the situation of Fig. 1.27, where we are going to allow preemption. Given that our system believes in ancestor worship, as we discussed above, we do not want a young whippersnapper preempting a venerable old sage, so Fig. 1.27(a) and not Fig. 1.27(b) is labeled *preempt*. We can now safely label Fig. 1.27(b) wait. This algorithm is known as *wound-wait*, because one transaction is supposedly wounded (it is actually killed) and the other waits. It is unlikely that this algorithm will make it to the Nomenclature Hall of Fame.



**Fig. 1.27. The wound-wait deadlock prevention algorithm.**

If an old process wants a resource held by a young one, the old process preempts the young one, whose transaction is then killed, as depicted in Fig. 1.27(a). The young one probably starts up again immediately, and tries to acquire the resource, leading to Fig. 1.27(b), forcing it to wait. Contrast this algorithm with wait-die. There, if an oldtimer wants a resource held by a young squirt, the oldtimer waits politely. However, if the young one wants a resource held by the old one, the young one is killed. It will undoubtedly start up again and be killed again. This cycle may go on many times before the old one releases the resource. Wound-wait does not have this nasty property.

### 1.3.6 Problems

1. Name at least three sources of delay that can be introduced between UTC broadcasting the time and the processors in a distributed system setting their internal clocks.
2. Consider the behavior of two machines in a distributed system. Both have clocks that are supposed to tick times per millisecond. One of them actually does, but the other ticks only 990 times per millisecond. If UTC updates come in once a minute, what is the maximum clock skew that will occur?
3. Suppose that two processes detect the demise of the coordinator simultaneously and both decide to hold an election using the bully algorithm. What happens?
4. Is optimistic concurrency control more or less restrictive than using timestamps? Why?
5. We have repeatedly said that when a transaction is aborted, the world is restored to its previous state, as though the transaction had never happened. We lied. Give an example where resetting the world is impossible.
6. A process with transaction timestamp 50 needs a resource held by a process with transaction timestamp 100. What happens in: (a) Wait-die? (b) Wound-wait?

## 1.4 SYSTEM MODELS

Processes run on processors. In a traditional system, there is only one processor, so the question of how the processor should be used does not come up. In a distributed system, with multiple processors, it is a major design issue. The processors in a distributed system can be organized in several ways. In this section we will look at two of the principal ones, the workstation model and the processor pool model, and a hybrid form encompassing features of each one. These models are rooted in fundamentally different philosophies of what a distributed is all about.

### 1.4.1 The Workstation Model

The workstation model is straightforward: the system consists of workstations (high-end personal computers) scattered throughout a building or campus and connected by a high-speed LAN. Some of the workstations may be in offices, and thus implicitly dedicated to a single user, whereas others may be in public areas and have several different users during the course of a day. In both cases, at any instant of time, a workstation either has a single user logged into it, and thus has an "owner" (however temporary), or it is idle.

The advantages of the workstation model are manifold and clear. The model is certainly easy to understand. Users have a fixed amount of dedicated computing power, and thus guaranteed response time. Sophisticated graphics programs can be very fast, since they can have direct access to the screen. Each user has a large degree of autonomy and can allocate his workstation's resources as he sees fit. Local disks add to this independence, and make it possible to continue working to a lesser or greater degree even in the face of file server crashes.

However, the model also has two problems. First, as processor chips continue to get cheaper, it will soon become economically feasible to give each user first 10 and later 100 CPUs. Having 100 workstations in your office makes it hard to see out the window. Second, much of the time users are not using their workstations, which are idle, while other users may need extra computing capacity and cannot get it. From a system-wide perspective, allocating resources in such a way that some users have resources they do not need while other users need these resources badly is inefficient.

The first problem can be addressed by making each workstation a personal multiprocessor. For example, each window on the screen can have a dedicated CPU to run its programs. Preliminary evidence from some early personal multiprocessors such as the DEC Firefly, suggest, however, that the mean number of CPUs utilized is rarely more than one, since users rarely have more than one active process at once. Again, this is an inefficient use of resources, but as get cheaper and cheaper as the technology improves, wasting them will become less of a sin.

### 1.4.2 Using Idle Workstations

The second problem, idle workstations, has been the subject of considerable research, primarily because many universities have a substantial number of personal workstations, some of which are idle. Measurements show that even at peak periods in the middle of the day, often as many as 30 percent of the workstations are idle at any given moment. In the evening, even more are idle. A variety of schemes have been proposed for using idle or otherwise underutilized workstations. We will describe the general principles behind this work in this section.

The earliest attempt to allow idle workstations to be utilized was the program that comes with Berkeley Unix. This program is called by *rsh* machine command in which the first argument names a machine and the second names a command to run on it. What *rsh* does is run the specified command on the specified machine. Although widely used, this program has several serious flaws. First, the user must tell which machine to use, putting the full burden of keeping track of idle machines on the user. Second, the program executes in the environment of the remote machine, which is usually different from the local environment. Finally, if someone should log into an idle machine on which a remote process is running, the process continues to run and the newly logged-in user either has to accept the lower performance or find another machine.

The research on idle workstations has centered on solving these problems. The key issues are:

1. How is an idle workstation found?
2. How can a remote process be run transparently?
3. What happens if the machine's owner comes back?

Let us consider these three issues, one at a time.

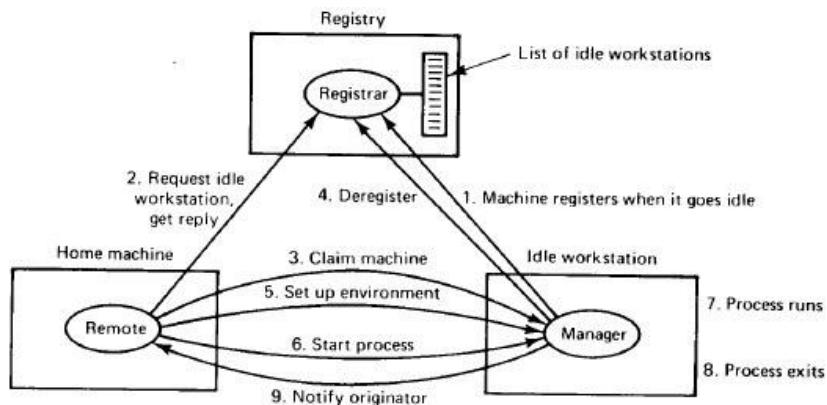
How is an idle workstation found? To start with, what is an idle workstation? At first glance, it might appear that a workstation with no one logged in at the console is an idle workstation, but with modern computer systems things

are not always that simple. In many systems, even with no one logged in there may be dozens of processes running, such as clock daemons, mail daemons, news daemons, and all manner of other daemons. On the other hand, a user who logs in when arriving at his desk in the morning, but otherwise does not touch the computer for hours, hardly puts any additional load on it. Different systems make different decisions as to what "idle" means, but typically, if no one has touched the keyboard or mouse for several minutes and no user-initiated processes are running, the workstation can be said to be idle. Consequently, there may be substantial differences in load between one idle workstation and another, due, for example, to the volume of mail coming into the first one but not the second.

The algorithms used to locate idle workstations can be divided into two categories: server driven and client driven. In the former, when a workstation goes idle, and thus becomes a potential compute server, it announces its availability. It can do this by entering its name, network address, and properties in a registry file (or data base), for example. Later, when a user wants to execute a command on an idle workstation, he types something like remote command and the remote program looks in the registry to find a suitable idle workstation. For reliability reasons, it is also possible to have multiple copies of the registry.

An alternative way for the newly idle workstation to announce the fact that it has become unemployed is to put a broadcast message onto the network. All other workstations then record this fact. In effect, each machine maintains its own private copy of the registry. The advantage of doing it this way is less overhead in finding an idle workstation and greater redundancy. The disadvantage is requiring all machines to do the work of maintaining the registry.

Whether there is one registry or many, there is a potential danger of race conditions occurring. If two users invoke the remote command simultaneously, and both of them discover that the same machine is idle, they may both try to start up processes there at the same time. To detect and avoid this situation, the remote program can check with the idle workstation, which, if still free, removes itself from the registry and gives the go-ahead sign. At this point, the caller can send over its environment and start the remote process, as shown in Fig. 1.28.



**Fig. 1.28. A registry-based algorithm for finding and using idle workstations.**

The other way to locate idle workstations is to use a client-driven approach. When remote is invoked, it broadcasts a request saying what program it wants to run, how much memory it needs, whether or not floating point is needed, and so on. These details are not needed if all the workstations are identical, but if the system is heterogeneous and not every program can run on every workstation, they are essential. When the replies come back, remote picks one and sets it up. One nice twist is to have "idle" workstations delay their responses slightly, with the delay being proportional to the current load. In this way, the reply from the least heavily loaded machine will come back first and be selected.

Finding a workstation is only the first step. Now the process has to be run there. Moving the code is easy. The trick is to set up the remote process so that it sees the same environment it would have locally, on the home workstation, and thus carries out the same computation it would have locally.

To start with, it needs the same view of the file system, the same working directory, and the same environment variables (shell variables), if any. After these have been set up, the program can begin running. The trouble starts when the first system call, say a READ, is executed. What should the kernel do? The answer depends very much on the system architecture. If the system is diskless, with all the files located on file servers, the kernel can just send the request to the appropriate file server, the same way the home machine would have done had the process

been running there. On the other hand, if the system has local disks, each with a complete file system, the request has to be forwarded back to the home machine for execution.

Some system calls must be forwarded back to the home machine no matter what, even if all the machines are diskless. For example, reads from the keyboard and writes to the screen can never be carried out on the remote machine. However, other system calls must be done remotely under all conditions. For example, the Unix system calls sbrk (adjust the size of the data segment), nice (set CPU scheduling priority), and profil (enable profiling of the program counter) cannot be executed on the home machine. In addition, all system calls that query the state of the machine have to be done on the machine on which the process is actually running. These include asking for the machine's name and network address, asking how much free memory it has, and so on.

System calls involving time are a problem because the clocks on different machines may not be synchronized. We saw how hard it is to achieve synchronization. Using the time on the remote machine may cause programs that depend on time, like make, to give incorrect results. Forwarding all time-related calls back to the home machine, however, introduces delay, which also causes problems with time.

To complicate matters further, certain special cases of calls which normally might have to be forwarded back, such as creating and writing to a temporary file, can be done much more efficiently on the remote machine. In addition, mouse tracking and signal propagation have to be thought out carefully as well. Programs that write directly to hardware devices, such as the screen's frame buffer, diskette, or magnetic tape, cannot be run remotely at all. All in all, making programs run on remote machines as though they were running on their home machines is possible, but it is a complex and tricky business.

The final question on our original list is what to do if the machine's owner comes back (i.e. somebody logs in or a previously inactive user touches the keyboard or mouse). The easiest thing is to do nothing, but this tends to defeat the idea of "personal" workstations. If other people can run programs on your workstation at the same time that you are trying to use it, there goes your guaranteed response.

Another possibility is to kill off the intruding process. The simplest way is to do this abruptly and without warning. The disadvantage of this strategy is that all work will be lost and the file system may be left in a chaotic state. A better way is to give the process fair warning, by sending it a signal to allow it to detect impending doom, and shut down gracefully (write edit buffers to the disk, close files, and so on). If it has not exited within a few seconds, it is then terminated. Of course, the program must be written to expect and handle this signal, something most existing programs definitely are not.

A completely different approach is to migrate the process to another machine, either back to the home machine or to yet another idle workstation. The hard part is not moving the user code and data, but finding and gathering up all the kernel data structures relating to the process that is leaving. For example, it may have open files, running timers, queued incoming messages, and other bits and pieces of information scattered around the kernel. These must all be carefully removed from the source machine and successfully reinstalled on the destination machine. There are no theoretical problems here, but the practical engineering difficulties are substantial.

In both cases, when the process is gone, it should leave the machine in the same state in which it found it, to avoid disturbing the owner. Among other items, this requirement means that not only must the process go, but also all its children and their children. In addition, mailboxes, network connections, and other system-wide data structures must be deleted, and some provision must be made to ignore RPC replies and other messages that arrive for the process after it is gone. If there is a local disk, temporary files must be deleted, and if possible, any files that had to be removed from its cache restored.

### **1.4.3 The Processor Pool Model**

Although using idle workstations adds a little computing power to the system, it does not address a more fundamental issue: What happens when it is feasible to provide 10 or 100 times as many CPUs as there are active users? One solution, as we saw, is to give everyone a personal multiprocessor. However this is a somewhat inefficient design.

An alternative approach is to construct a *processor pool*, a rack full of CPUs in the machine room, which can be dynamically allocated to users on demand. Instead of giving users personal workstations, in this model they are given high-performance graphics terminals. This idea is based on the observation that what many users really want is a high-quality graphical interface and good performance. Conceptually, it is much closer to traditional

timesharing than to the personal computer model, although it is built with modern technology (low-cost microprocessors).

The motivation for the processor pool idea comes from taking the diskless workstation idea a step further. If the file system can be centralized in a small number of file servers to gain economies of scale, it should be possible to do the same thing for compute servers. By putting all the CPUs in a big rack in the machine room, power supply and other packaging costs can be reduced, giving more computing power for a given amount of money. Furthermore, it permits the use of cheaper X terminals (or even ordinary ASCII terminals), and the number of users from the number of workstations. The model also allows for easy incremental growth. If the computing load increases by 10 percent, you can just buy 10 percent more processors and put them in the pool.

In effect, we are converting all the computing power into "idle workstations" that can be accessed dynamically. Users can be assigned as many CPUs as they need for short periods, after which they are returned to the pool so that other users can have them. There is no concept of ownership here: all the processors belong equally to everyone.

The biggest argument for centralizing the computing power in a processor pool comes from queueing theory. A queueing system is a situation in which users generate random requests for work from a server. When the server is busy, the users queue for service and are processed in turn. Common examples of queueing systems are bakeries, airport check-in counters, supermarket check-out counters, and numerous others.

Queueing systems are useful because it is possible to model them analytically. Let us call the total input rate  $l$  requests per second, from all the users combined. Let us call the rate at which the server can process requests  $m$ . For stable operation, we must have  $m > l$ . If the server can handle 100 requests/sec, but the users continuously generate 110 requests/sec, the queue will grow without bound. (Small intervals in which the input rate exceeds the service rate are acceptable, provided that the mean input rate is lower than the service rate and there is enough buffer space.)

It can be proven that the mean time between issuing a request and getting a complete response,  $T$ , is related to  $l$  and  $m$  by the formula  $T = 1/(m - l)$ .

Suppose that we have  $n$  personal multiprocessors, each with some number of and each one forms a separate queueing system with request arrival rate  $l$  and CPU processing rate  $m$ . The mean response time,  $T$ , will be as given above. Now consider what happens if we scoop up all the CPUs and place them in a single processor pool. Instead of having  $n$  small queueing systems running in parallel, we now have one large one, with an input rate  $nl$  and a service rate  $nm$ . Let us call the mean response time of this combined system  $T_1$ . From the formula above we find  $T_1 = 1/(nl - nm) = T/n$ . This surprising result says that by replacing  $n$  small resources by one big one that is  $n$  times more powerful, we can reduce the average response time  $n$ -fold.

This result is extremely general and applies to a large variety of systems. The effect arises because dividing the processing power into small servers (e.g. personal workstations), each with one user, is a poor match to a workload of randomly arriving requests. Much of the time, a few servers are busy, even overloaded, but most are idle. It is this wasted time that is eliminated in the processor pool model, and the reason why it gives better overall performance. The concept of using idle workstations is a weak attempt at recapturing the wasted cycles, but it is complicated and has many problems, as we have seen.

In fact, this queueing theory result is one of the main arguments against having distributed systems at all. Given a choice between one centralized MIPS CPU and 100 private, dedicated, 10-MIPS CPUs, the mean response time of the former will be 100 times better, because no cycles are ever wasted. The machine goes idle only when no user has any work to do. This fact argues in favor of concentrating the computing power as much as possible.

However, mean response time is not everything. There are also arguments in favor of distributed computing, such as cost. If a single 1000-MIPS CPU is much more expensive than 100 10-MIPS the latter may be much better. It may not even be possible to build such a large machine at any price. Reliability and fault tolerance are also factors.

Also, personal workstations have a uniform response, independent of what other people are doing (except when the network or file servers are jammed). For some users, a low variance in response time may be perceived as more important than the mean response time itself. Consider, for example, editing on a private workstation on

which asking for the next page to be displayed always takes 500 msec. Now consider editing on a large, centralized, shared computer on which asking for the next page takes 5 msec 95 percent of the time and 5 one time in 20. Even though the mean here is twice as good as on the workstation, the users may consider the performance intolerable. On the other hand, to the user with a huge simulation to run, the big computer may win hands down.

We have tacitly assumed that a pool of  $n$  processors is effectively the same thing as a single processor that is  $n$  times as fast as a single processor. In reality, this assumption is justified only if all requests can be split up in such a way as to allow them to run on all the processors in parallel. If a job can be split into, say, only 5 parts, then the processor pool model has an effective service time only 5 times better than that of a single processor.

Still, the processor pool model is a much cleaner way of getting extra computing power than looking around for idle workstations and sneaking over there while nobody is looking. By starting out with the assumption that no processor belongs to anyone, we get a design based on the concept of requesting machines from the pool, using them, and putting them back when done. There is also no need to forward anything back to a "home" machine because there are none. There is also no danger of the owner coming back, because there are no owners.

In the end, it all comes down to the nature of the workload. If all people are doing is simple editing and occasionally sending an electronic mail message or two, having a personal workstation is probably enough. If, on the other hand, the users are engaged in a large software development project, frequently running make on large directories, or are trying to invert massive sparse matrices, or do major simulations or run big artificial intelligence or VLSI routing programs, constantly hunting for substantial numbers of idle workstations will be no fun at all. In all these situations, the processor pool idea is fundamentally much simpler and more attractive.

#### **1.4.4 A Hybrid Model**

A possible compromise is to provide each user with a personal workstation and to have a processor pool in addition. Although this solution is more expensive than either a pure workstation model or a pure processor pool model, it combines the advantages of both of the others.

Interactive work can be done on workstations, giving guaranteed response. Idle workstations, however, are not utilized, making for a simpler system design. They are just left unused. Instead, all noninteractive processes run on the processor pool, as does all heavy computing in general. This model provides fast interactive response, an efficient use of resources, and a simple design.

#### **1.4.5 Fault Tolerance**

A system is said to fail when it does not meet its specification. In some cases, such as a supermarket's distributed ordering system, a failure may result in some store running out of canned beans. In other cases, such as in a distributed air traffic control system, a failure may be catastrophic. As computers and distributed systems become widely used in safety-critical missions, the need to prevent failures becomes correspondingly greater. In this section we will examine some issues concerning system failures and how they can be avoided.

##### **1.4.5.1 Component Faults**

Computer systems can fail due to a fault in some component, such as a processor, memory, device, cable, or software. A *fault* is a malfunction, possibly caused by a design error, a manufacturing error, a programming error, physical damage, deterioration in the course of time, harsh environmental conditions (it snowed on the computer), unexpected inputs, operator error, rodents eating part of it, and many other causes. Not all faults lead (immediately) to system failures, but some do.

Faults are generally classified as transient, intermittent, or permanent. *Transient faults* occur once and then disappear. If the operation is repeated, the fault goes away. A bird flying through the beam of a microwave transmitter may cause lost bits on some network (not to mention a roasted bird). If the transmission times out and is retried, it will probably work the second time.

An *intermittent fault* occurs, then vanishes of its own accord, then reappears, and so on. A loose contact on a connector will often cause an intermittent fault. Intermittent faults cause a great deal of aggravation because they are difficult to diagnose. Typically, whenever the fault doctor shows up, the system works perfectly.

A *permanent fault* is one that continues to exist until the faulty component is repaired. Burnt-out chips, software bugs, and disk head crashes often cause permanent faults.

The goal of designing and building fault-tolerant systems is to ensure that the system as a whole continues to function correctly, even in the presence of faults. This aim is quite different from simply engineering the individual components to be highly reliable, but allowing (even expecting) the system to fail if one of the components does so.

Faults and failures can occur at all levels: transistors, chips, boards, processors, operating systems, user programs, and so on. Traditional work in the area of fault tolerance has been concerned mostly with the statistical analysis of electronic component faults. Very briefly, if some component has a probability  $p$  of malfunctioning in a given second of time, the probability of it not failing for  $k$  consecutive seconds and then failing is  $p(1-p)^k$ . The expected time to failure is then given by the formula

$$\text{mean time to failure} = \sum(kp(1-p)^{k-1}, k=1..\infty)$$

Using the well-known equation for an infinite sum starting at  $k=1$ :  $\sum(a^k)=a/(1-a)$ , substituting  $a=1-p$ , differentiating both sides of the resulting equation with respect to  $p$ , and multiplying through by  $-p$  we see that

$$\text{mean time to failure} = 1/p$$

For example, if the probability of a crash is  $10^{-6}$  per second, the mean time to failure is  $10^6$  sec or about 1.6 days.

#### 1.4.5.2 System Failures

In a critical distributed system, often we are interested in making the system be able to survive component (in particular, processor) faults, rather than just making these unlikely. System reliability is especially important in a distributed system due to the large number of components present, hence the greater chance of one of them being faulty.

For the rest of this section, we will discuss processor faults or crashes, but this should be understood to mean equally well process faults or crashes due to software bugs). Two types of processor faults can be distinguished:

1. Fail-silent faults.
2. Byzantine faults.

With *fail-silent faults*, a faulty processor just stops and does not respond to subsequent input or produce further output, except perhaps to announce that it is no longer functioning. These are also called *fail-stop* faults. With *Byzantine faults*, a faulty processor continues to run, issuing wrong answers to questions, and possibly working together maliciously with other faulty processors to give the impression that they are all working correctly when they are not. Undetected software bugs often exhibit Byzantine faults. Clearly, dealing with Byzantine faults is going to be much more difficult than dealing with fail-silent ones.

The term "Byzantine" refers to the Byzantine Empire, a time (330-1453) and place (the Balkans and modern Turkey) in which endless conspiracies, intrigue, and untruthfulness were alleged to be common in ruling circles.

Combinations of these faults with communication line faults were also considered, but since standard protocols can recover from line errors in predictable ways, we will examine only processor faults.

#### 1.4.5.3 Synchronous versus Asynchronous Systems

As we have just seen, component faults can be transient, intermittent, or permanent, and system failures can be fail-silent or Byzantine. A third orthogonal axis deals with performance in a certain abstract sense. Suppose that we have a system in which if one processor sends a message to another, it is guaranteed to get a reply within a time  $T$  known in advance. Failure to get a reply means that the receiving system has crashed. The time  $T$  includes sufficient time to deal with lost messages (by sending them up to  $n$  times).

In the context of research on fault tolerance, a system that has the property of always responding to a message within a known finite bound if it is working is said to be *synchronous*. A system not having this property is said to be *asynchronous*. While this terminology is unfortunately, since it conflicts with more traditional uses of the terms, it is widely used among workers in fault tolerance.

It should be intuitively clear that asynchronous systems are going to be harder to deal with than synchronous ones. If a processor can send a message and know that the absence of a reply within  $T$  sec means that the intended

recipient has failed, it can take corrective action. If there is no upper limit to how long the response might take, even determining whether there has been a failure is going to be a problem.

#### 1.4.5.4 Use of Redundancy

The general approach to fault tolerance is to use redundancy. Three kinds are possible: information redundancy, time redundancy, and physical redundancy. With information redundancy, extra bits are added to allow recovery from garbled bits. For example, a Hamming code can be added to transmitted data to recover from noise on the transmission line.

With time redundancy, an action is performed, and then, if need be, it is performed again. Using the atomic transactions is an example of this approach. If a transaction aborts, it can be redone with no harm. Time redundancy is especially helpful when the faults are transient or intermittent.

With physical redundancy, extra equipment is added to make it possible for the system as a whole to tolerate the loss or malfunctioning of some components. For example, extra processors can be added to the system so that if a few of them crash, the system can still function correctly.

There are two ways to organize these extra processors: active replication and primary backup. Consider the case of a server. When active replication is used, all the processors are used all the time as servers (in parallel) in order to hide faults completely. In contrast, the primary backup scheme just uses one processor as a server, replacing it with a backup if it fails.

We will discuss these two strategies below. For both of them, the issues are:

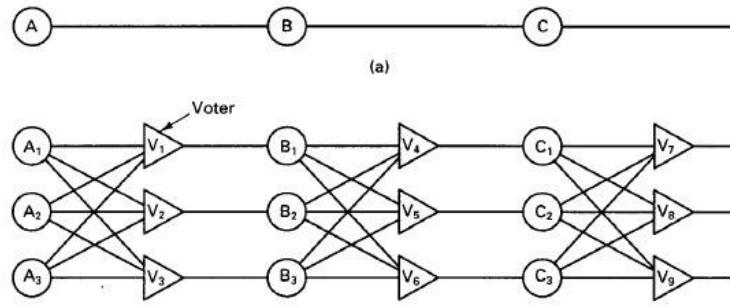
1. The degree of replication required.
2. The average and worst-case performance in the absence of faults.
3. The average and worst-case performance when a fault occurs.

Theoretical analyses of many fault-tolerant systems can be done in these terms.

#### 1.4.5.5 Fault Tolerance Using Active Replication

Active replication is a well-known technique for providing fault tolerance using physical redundancy. It is used in biology (mammals have two eyes, two ears, two lungs, etc.), aircraft (747s have four engines but can fly on three), and sports (multiple referees in case one misses an event). Some authors refer to active replication as the state machine approach.

It has also been used for fault tolerance in electronic circuits for years. Consider, for example, the circuit of Fig. 1.29(a). Here signals pass through devices A, B and C, in sequence. If one of them is faulty, the final result will probably be wrong. In Fig. 1.29(b), each device is replicated three times. Following each stage in the circuit is a triplicated voter. Each voter is a circuit that has three inputs and one output. If two or three of the inputs are the same, the output is equal to that input. If all three inputs are different, the output is undefined. This kind of design is known as TMR (Triple Modular Redundancy).



**Fig. 1.29. Triple modular redundancy.**

Suppose element A<sub>2</sub> fails. Each of the voters, V<sub>1</sub>, V<sub>2</sub> and V<sub>3</sub> gets two good (identical) inputs and one rogue input, and each of them outputs the correct value to the second stage. In essence, the effect of A<sub>2</sub> failing is completely masked, so that the inputs to B<sub>1</sub>, B<sub>2</sub> and B<sub>3</sub> are exactly the same as they would have been had no fault occurred.

Now consider what happens if B<sub>3</sub> and C<sub>1</sub> are also faulty, in addition to A<sub>2</sub>. These effects are also masked, so the three final outputs are still correct.

At first it may not be obvious why three voters are needed at each stage. After all, one voter could also detect and pass though the majority view. However, a voter is also a component and can also be faulty. Suppose, for example, that  $V_1$  malfunctions. The input to  $B_1$  will then be wrong, but as long as everything else works,  $B_2$  and  $B_3$  will produce the same output and  $V_4$ ,  $V_5$ , and  $V_6$  will all produce the correct result into stage three. A fault in  $V_1$  is effectively no different than a fault in  $B_1$  produces incorrect output, but in both cases it is voted down later.

Although not all fault-tolerant distributed operating systems use TMR, the technique is very general, and should give a clear feeling for what a tolerant system is, as opposed to a system whose individual components are highly reliable but whose organization is not fault tolerant. Of course, TMR can be applied recursively, for example, to make a chip highly reliable by using TMR inside it, unknown to the designers who use the chip.

Getting back to fault tolerance in general and active replication in particular, in many systems, servers act like big finite-state machines: they accept requests and produce replies. Read requests do not change the state of the server, but write requests do. If each client request is sent to each server, and they all are received and processed in the same order, then after processing each one, all nonfaulty servers will be in exactly the same state and will give the same replies. The client or voter can combine all the results to mask faults.

An important issue is how much replication is needed. The answer depends on the amount of fault tolerance desired. A system is said to be  $k$  fault tolerant if it can survive faults in  $k$  components and still meet its specifications. If the components, say processors, fail silently, then having  $k + 1$  of them is enough to provide  $k$  fault tolerance. If  $k$  of them simply stop, then the answer from the other one can be used.

On the other hand, if the processors exhibit Byzantine failures, continuing to run when sick and sending out erroneous or random replies, a minimum of  $2k + 1$  processors are needed to achieve  $k$  fault tolerance. In the worst case, the  $k$  failing processors could accidentally (or even intentionally) generate the same reply. However, the remaining  $k + 1$  will also produce the same answer, so the client or voter can just believe the majority.

Of course, in theory it is fine to say that a system is  $k$  fault tolerant and just let the  $k + 1$  identical replies outvote the  $k$  identical replies, but in practice it is hard to imagine circumstances in which one can say with certainty that  $k$  processors can fail but  $k + 1$  processors cannot fail. Thus even in a fault-tolerant system some kind of statistical analysis may be needed.

An implicit precondition for this finite state machine model to be relevant is that all requests arrive at all servers in the same order, sometimes called the *atomic broadcast problem*. Actually, this condition can be relaxed slightly, since reads do not matter and some writes may commute, but the general problem remains. One way to make sure that all requests are processed in the same order at all servers is to number them globally. Various protocols have been devised to accomplish this goal. For example, all requests could first be sent to a global number server to get a serial number, but then provision would have to be made for the failure of this server (e.g. by making it internally fault tolerant).

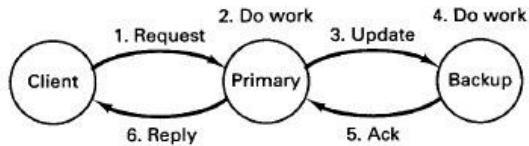
Another possibility is to use Lamport's logical clocks. If each message sent to a server is tagged with a timestamp, and servers process all requests in timestamp order, all requests will be processed in the same order at all servers. The trouble with this method is that when a server receives a request, it does not know whether any earlier requests are currently under way. In fact, most timestamp solutions suffer from this problem. In short, active replication is not a trivial matter.

#### 1.4.5.6 Fault Tolerance Using Primary Backup

The essential idea of the primary-backup method is that at any one instant, one server is the primary and does all the work. If the primary fails, the backup takes over. Ideally, the switch should take place in a clean way and be noticed only by the client operating system, not by the application programs. Like active replication, this scheme is widely used in the world. Some examples are government (the Vice President), aviation (co-pilots), automobiles (spare tires), and diesel-powered electrical generators in hospital operating rooms.

Primary-backup fault tolerance has two major advantages over active replication. First, it is simpler during normal operation since messages go to just one server (the primary) and not to a whole group. The problems associated with ordering these messages also disappear. Second, in practice it requires fewer machines, because at any instant one primary and one backup is needed (although when a backup is put into service as a primary, a new backup is needed instantly). On the downside, it works poorly in the presence of Byzantine failures in which the primary

erroneously claims to be working perfectly. Also, recovery from a primary failure can be complex and time consuming.



**Fig. 1.30. A simple primary-backup protocol on a write operation.**

As an example of the primary-backup solution, consider the simple protocol of Fig. 1.30 in which a write operation is depicted. The client sends a message to the primary, which does the work and then sends an update message to the backup. When the backup gets the message, it does the work and then sends an acknowledgement back to the primary. When the acknowledgement arrives, the primary sends the reply to the client.

Now let us consider the effect of a primary crash at various moments during an RPC. If the primary crashes before doing the work (step 2), no harm is done. The client will time out and retry. If it tries often enough, it will eventually get the backup and the work will be done exactly once. If the primary crashes after doing the work but before sending the update, when the backup takes over and the request comes in again, the work will be done a second time. If the work has side effects, this could be a problem. If the primary crashes after step 4 but before step 6, the work may end up being done three times, once by the primary, once by the backup as a result of step 3, and once after the backup becomes the primary. If requests carry identifiers, it may be possible to ensure that the work is done only twice, but getting it done exactly once is difficult to impossible.

One theoretical and practical problem with the primary-backup approach is when to cut over from the primary to the backup. In the protocol above, the backup could send: "Are you alive?" messages periodically to the primary. If the primary fails to respond within a certain time, the backup would take over.

However, what happens if the primary has not crashed, but is merely slow (we have an asynchronous system)? There is no way to distinguish between a slow primary and one that has gone down. Yet there is a need to make sure that when the backup takes over, the primary really stops trying to act like the primary. Ideally the backup and primary should have a protocol to discuss this, but it is hard to negotiate with the dead. The best solution is a hardware mechanism in which the backup can forcibly stop or reboot the primary. Note that all primary-backup schemes require agreement, which is tricky to achieve, whereas active replication does not always require an agreement protocol (e.g. TMR).

A variant of the approach of Fig. 1.30 uses a dual-ported disk shared between the primary and secondary. In this configuration, when the primary gets a request, it writes the request to disk before doing any work and also writes the results to disk. No messages to or from the backup are needed. If the primary crashes, the backup can see the state of the world by reading the disk. The disadvantage of this scheme is that there is only one disk, so if that fails, everything is lost. Of course, at the cost of extra equipment and performance, the disk could also be replicated and all writes could be done to both disks.

#### 1.4.5.7 Agreement in Faulty Systems

In many distributed systems there is a need to have processes agree on something. Examples are electing a coordinator, deciding whether to commit a transaction or not, dividing up tasks among workers, synchronization, and so on. When the communication and processors are all perfect, reaching such agreement is often straightforward, but when they are not, problems arise. In this section we will look at some of the problems and their solutions (or lack thereof).

The general goal of distributed agreement algorithms is to have all the faulty processors reach consensus on some issue, and do that within a finite number of steps. Different cases are possible depending on system parameters, including:

1. Are messages delivered reliably all the time?
2. Can processes crash, and if so, fail-silent or Byzantine?
3. Is the system synchronous or asynchronous?

Before considering the case of faulty processors, let us look at the "easy" case of perfect processors but communication lines that can lose messages. There is a famous problem, known as the *two-army problem*, which illustrates the difficulty of getting even two perfect processors to reach agreement about 1 bit of information. The

red army, with 5000 troops, is encamped in a valley. Two blue armies, each 3000 strong, are encamped on the surrounding hillsides overlooking the valley. If the two blue armies can coordinate their attacks on the red army, they will be victorious. However, if either one attacks by itself it will be slaughtered. The goal of the blue armies is to reach agreement about attacking. The catch is that they can only communicate using an unreliable channel: sending a messenger who is subject to capture by the red army.

Suppose that the commander of blue army 1, General Alexander, sends a message to the commander of blue army 2, General Bonaparte, reading: "I have a plan-let's attack at dawn tomorrow." The messenger gets through and Bonaparte sends him back with a note saying: "Splendid idea, Alex. See you at dawn tomorrow." The messenger gets back to his base safely, delivers his messages, and Alexander tells his troops to prepare for battle at dawn.

However, later that day, Alexander realizes that Bonaparte does not know if the messenger got back safely and not knowing this, may not dare to attack. Consequently, Alexander tells the messenger to go tell Bonaparte that his (Bonaparte's) message arrived and that the battle is set.

Once again the messenger gets through and delivers the acknowledgement. But now Bonaparte worries that Alexander does not know if the acknowledgement got through. He reasons that if Bonaparte thinks that the messenger was captured, he will not be sure about his (Alexander's) plans, and may not risk the attack, so he sends the messenger back again.

Even if the messenger makes it through every time, it is easy to show that Alexander and Bonaparte will never reach agreement, no matter how many acknowledgements they send. Assume that there is some protocol that terminates in a finite number of steps. Remove any extra steps at the end to get the minimum protocol that works. Some message is now the last one and it is essential to the agreement (because this is the minimum protocol). If this message fails to arrive, the war is off.

However, the sender of the last message does not know if the last message arrived. If it did not, the protocol did not complete and the other general will not attack. Thus the sender of the last message cannot know if the war is scheduled or not, and hence cannot safely commit his troops. Since the receiver of the last message knows the sender cannot be sure, he will not risk certain death either, and there is no agreement. Even with nonfaulty processors (generals), agreement between even two processes is not possible in the face of unreliable communication.

Now let us assume that the communication is perfect but the processors are not. The classical problem here also occurs in a military setting and is called the *Byzantine generals problem*. In this problem the red army is still encamped in the valley, but  $n$  blue generals all head armies on the nearby hills. Communication is done by telephone and is perfect, but  $m$  of the generals are traitors (faulty) and are actively trying to prevent the loyal generals from reaching agreement by feeding them incorrect and contradictory information (to model malfunctioning processors). The question is now whether the loyal generals can still reach agreement.

For the sake of generality, we will define agreement in a slightly different way here. Each general is assumed to know how many troops he has. The goal of the problem is for the generals to exchange troop strengths, so that at the end of the algorithm, each general has a vector of length  $n$  corresponding to all thearmies. If general  $i$  is loyal, then element  $i$  is his troop strength; otherwise, it is undefined.

A recursive algorithm was devised by Lamport (1982) that solves this problem under certain conditions. In Fig. 1.31 we illustrate the working of the algorithm for the case of  $n = 4$  and  $m = 1$ . For these parameters, the algorithm operates in four steps. In step one, every general sends a (reliable) message to every other general announcing his truth strength. Loyal generals tell the truth; traitors may tell every other general a different lie. In Fig. 1.31(a) we see that general 1 reports 1K troops, general 2 reports 2K troops, general 3 lies to everyone, giving  $x$ ,  $y$ , and  $z$ , respectively, and general 4 reports 4K troops. In step 2, the results of the announcements of step 1 are collected together in the form of the vectors of Fig. 1.31(b).

Step 3 consists of every general passing his vector from Fig. 1.31(b) to every other general. Here, too, general 3 lies through his teeth, inventing 12 new values,  $a$  through  $l$ . The results of step 3 are shown in Fig. 1.31(c). Finally, in step 4, each general examines the  $i$ th element of each of the newly received vectors. If any value has a majority, that value is put into the result vector. If no value has a majority, the corresponding element of the result vector is marked UNKNOWN. From Fig. 1.31(c) we see that generals 1, 2, and 4 all come to agreement on (1, 2, UNKNOWN, 4) which is the correct result. The traitor was not able to gum up the works.

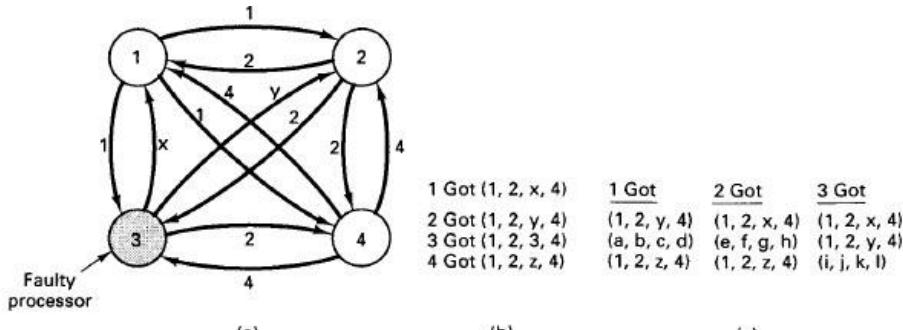


Fig. 1.31. The Byzantine generals problem for 3 loyal generals and traitor. (a) The generals announce their troop strengths (in units of 1, 2, 4, x, y). (b) The vectors that each general assembles based on (a). (c) The vectors that each general receives in step 2.

Now let us revisit this problem for  $n = 3$  and  $m = 1$ , that is, only two loyal generals and one traitor, as illustrated in Fig. 1.32. Here we see that in Fig. 1.32(c) neither of the loyal generals sees a majority for element 1, element 2, or element 3, so all of them are marked UNKNOWN. The algorithm has failed to produce agreement.

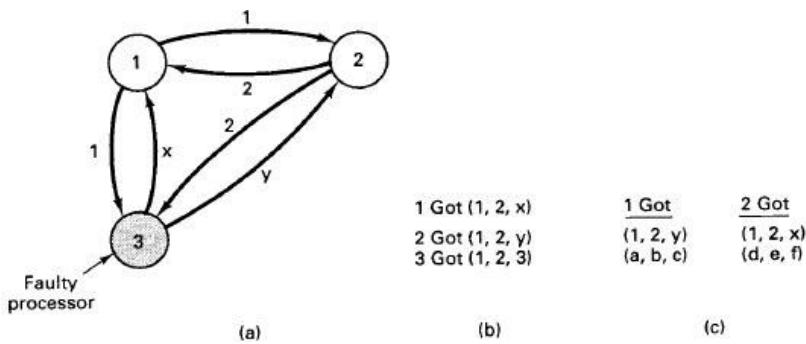


Fig. 1.32. The same as Fig. 1.31, except now with 2 loyal generals and one traitor.

Lamport proved that in a system with  $m$  faulty processors, agreement can be achieved only if  $2m + 1$  correctly functioning processors are present, for a total of  $3m + 1$ . Put in slightly different terms, agreement is possible only if more than two-thirds of the processors are working properly.

Worse yet, later on Fischer proved that in a distributed system with asynchronous processors and unbounded transmission delays, no agreement is possible if even one processor is faulty (even if that one processor fails silently). The problem with asynchronous systems is that arbitrarily slow processors are indistinguishable from dead ones. Many other theoretical results are known about when agreement is possible and when it is not.

## 1.4.6 Real-Time Distributed Systems

Fault-tolerant systems are not the only kind of specialized distributed systems. The real-time systems form another category. Sometimes these two are combined to give fault-tolerant real-time systems. In this section we will examine various aspects of real-time distributed systems.

### 1.4.6.1 What Is a Real-Time System?

For most programs, correctness depends only on the logical sequence of instructions executed, not when they are executed. If a C program correctly computes the double-precision floating-point square root function on a 200-MHz engineering workstation, it will also compute the function correctly on a MHz 8088-based personal computer, only slower.

In contrast, real-time programs (and systems) interact with the external world in a way that involves time. When a stimulus appears, the system must respond to it in a certain way and before a certain deadline. If it delivers the correct answer, but after the deadline, the system is regarded as having failed. When the answer is produced is as important as which answer is produced.

Consider a simple example. An audio compact disk player consists of a CPU that takes the bits arriving from the disk and processes them to generate music. Suppose that the CPU is just barely fast enough to do the job. Now imagine that a competitor decides to build a cheaper player using a CPU running at one-third the speed. If it

buffers all the incoming bits and plays them back at one-third the expected speed, people will wince at the sound, and if it only plays every third note, the audience will not be wildly ecstatic either. Unlike the earlier square root example, time is inherently part of the specification of correctness here.

Many other applications involving the external world are also inherently real time. Examples include computers embedded in television sets and video recorders, computers controlling aircraft ailerons and other parts (so called fly-by-wire), automobile subsystems controlled by computers (drive-by-wire?), military computers controlling guided antitank missiles (shoot-by-wire?), computerized air traffic control systems, scientific experiments ranging from particle accelerators to psychology lab mice with electrodes in their brains, automated factories, telephone switches, robots, medical intensive care units, CAT scanners, automatic stock trading systems, and others.

Many real-time applications and systems are highly structured, much more so than general-purpose distributed systems. Typically, an external device (possibly a clock) generates a stimulus for the computer, which must then perform certain actions before a deadline. When the required work has been completed, the system becomes idle until the next stimulus arrives.

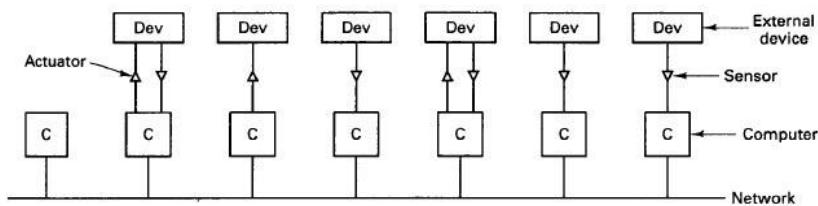
Frequently, the stimuli are *periodic*, with a stimulus occurring regularly every AT seconds, such as a computer in a TV set or VCR getting a new frame every 1/60 of a second. Sometimes stimuli are *aperiodic*, meaning that they are recurrent, but not regular, as in the arrival of an aircraft in a air traffic controller's air space. Finally, some stimuli are *sporadic* (unexpected), such as a device overheating.

Even in a largely periodic system, a complication is that there may be many types of events, such as video input, audio input, and motor drive management, each with its own period and required actions.

Despite the fact that the CPU may have to deal with multiple event streams, it is not acceptable for it to say: It is true that I missed event B, but it is not my fault-I was still working on A when B happened. While it is not hard to manage two or three input streams with priority interrupts, as applications get larger and more complex (e.g. automated factory assembly lines with thousands of robots), it will become more and more difficult for one machine to meet all the deadlines and other real-time constraints.

Consequently, some designers are experimenting with the idea of putting a dedicated microprocessor in front of each real-time device to accept output from it whenever it has something to say, and give it input at whatever speed it requires. Of course, this does not make the real-time character go away, but instead gives rise to a distributed real-time system, with its own unique characteristics and challenges (e.g. real-time communication).

Distributed real-time systems can often be structured as illustrated in Fig. 1.33. Here we see a collection of computers connected by a network. Some of these are connected to external devices that produce or accept data or expect to be controlled in real time. The computers may be tiny microcontrollers built into the devices, or stand-alone machines. In both cases they usually have sensors for receiving signals from the devices and/or actuators for sending signals to them. The sensors and actuators may be digital or analog.



**Fig. 1.33 A distributed real-time computer system.**

Real-time systems are generally split into two types depending on how serious their deadlines are and the consequences of missing one. These are:

1. Soft real-time systems.
2. Hard real-time systems.

*Soft real-time* means that missing an occasional deadline is all right. For example, a telephone switch might be permitted to lose or misroute one call in under overload conditions and still be within specification. In contrast, even a single missed deadline in a hard real-time system is unacceptable, as this might lead to loss of life or an environmental catastrophe. In practice, there are also intermediate systems where missing a deadline means you have to kill off the current activity, but the consequence is not fatal. For example, if a soda bottle on a conveyor

belt has passed by the nozzle, there is no point in continuing to squirt soda at it, but the results are not fatal. Also, in some real-time systems, some subsystems are hard real time whereas others are soft real time.

### 1.4.6.2 Design Issues

Real-time distributed systems have some unique design issues. In this section we will examine some of the most important ones.

**Clock Synchronization.** The first issue is the maintenance of time itself. With multiple computers, each having its own local clock, keeping the clocks in synchrony is a key issue.

**Event-Triggered versus Time-Triggered Systems.** In an event-triggered real-time system, when a significant event in the outside world happens, it is detected by some sensor, which then causes the attached CPU to get an interrupt. Event-triggered systems are thus interrupt driven. Most real-time systems work this way. For soft real-time systems with lots of computing power to spare, this approach is simple, works well, and is still widely used. Even for more complex systems, it works well if the compiler can analyze the program and know all there is to know about the system behavior once an event happens, even if it cannot tell when the event will happen.

The main problem with event-triggered systems is that they can fail under conditions of heavy load, that is, when many events are happening at once. Consider, for example, what happens when a pipe ruptures in a controlled nuclear reactor. Temperature alarms, pressure alarms, radioactivity alarms, and other alarms will all go off at once, causing massive interrupts. This *event shower* may overwhelm the computing system and bring it down, potentially causing problems far more serious than the rupture of a single pipe.

An alternative design that does not suffer from this problem is the triggered real-time system. In this kind of system, a clock interrupt occurs every  $dT$  milliseconds. At each clock tick (selected) sensors are sampled and (certain) actuators are driven. No interrupts occur other than clock ticks.

As an example of the difference between these two approaches, consider the design of an elevator controller in a 100-story building. Suppose that the elevator is sitting peacefully on the 60th floor waiting for customers. Then someone pushes the call button on the first floor. Just 100 msec later, someone else pushes the call button on the 100<sup>th</sup> floor. In an event-triggered system, the first call generates an interrupt, which causes the elevator to take off downward. The second call comes in after the decision to go down has already been made, so it is noted for future reference, but the elevator continues on down.

Now consider a time-triggered elevator controller that samples every 500 msec. If both calls fall within one sampling period, the controller will have to make a decision, for example, using the nearest-customer-first rule, in which case it will go up.

In summary, event-triggered designs give faster response at low load but more overhead and chance of failure at high load. Time-trigger designs have the opposite properties and are furthermore only suitable in a relatively static environment in which a great deal is known about system behavior in advance. Which one is better depends on the application. In any event, we note that there is much lively controversy over this subject in real-time circles.

**Predictability.** One of the most important properties of any real-time system is that its behavior be predictable. Ideally, it should be clear at design time that the system can meet all of its deadlines, even at peak load. Statistical analyses of behavior assuming independent events are often misleading because there may be unsuspected correlations between events, as between the temperature, pressure, and radioactivity alarms in the ruptured pipe example above. Most distributed system designers are used to thinking in terms of independent users accessing shared files at random or numerous travel agents accessing a shared airline data base at unpredictable times. Fortunately, this kind of chance behavior rarely holds in a real-time system. More often, it is known that when event E is detected, process X should be run, followed by Y and Z, in either order or in parallel. Furthermore, it is often known (or should be known) what the worst-case behavior of these processes is. For example, if it is known that X needs 50 msec, Y and Z need 60 msec each, and process startup takes 5 msec, then it can be guaranteed in advance that the system can flawlessly handle five periodic type E events per second in the absence of any other work. This kind of reasoning and modeling leads to a deterministic rather than a stochastic system.

**Fault Tolerance.** Many real-time systems control safety-critical devices in vehicles, hospitals, and power plants, so fault tolerance is frequently an issue. Active replication is sometimes used, but only if it can be done without extensive (and thus consuming) protocols to get everyone to agree on everything all the time. Primary-backup

schemes are less popular because deadlines may be missed during after the primary fails. A hybrid approach is to follow the leader, in which one machine makes all the decisions, but the others just do what it says to do without discussion, ready to take over at a moment's notice.

In a safety-critical system, it is especially important that the system be able to handle the worst-case scenario. It is not enough to say that the probability of three components failing at once is so low that it can be ignored. Failures are not always independent. For example, during a sudden electric power failure, everyone grabs the telephone, possibly causing the phone system to overload, even though it has its own independent power generation system. The peak load on the system often occurs precisely at the moment when the maximum number of components have failed because much of the traffic is related to reporting the failures. Consequently, fault-tolerant real-time systems must be able to cope with the maximum number of faults and the maximum load at the same time.

Some real-time systems have the property that they can be stopped cold when a serious failure occurs. For instance, when a railroad signaling system unexpectedly blacks out, it may be possible for the control system to tell every train to stop immediately. If the system design always spaces trains far enough apart and all trains start braking more-or-less simultaneously, it will be possible to avert disaster and the system can recover gradually when the power comes back on. A system that can halt operation like this without danger is said to be *fail-safe*.

**Language Support.** While many real-time systems and applications are programmed in languages such as C, specialized real-time languages can potentially be of great assistance. For example, in such a language, it should be easy to express the work as a collection of short tasks (e.g. threads) that can be scheduled independently, subject to user-defined precedence and mutual exclusion constraints. The language should be designed so that the maximum execution time of every task can be computed at compile time. This requirement means that the language cannot support general while loops. Iteration must be done using for loops with constant parameters. Recursion cannot be tolerated either. Even these restrictions may not be enough to make it possible to calculate the execution time of each task in advance since cache misses, page faults, and cycle stealing by DMA channels all affect performance, but they are a start. Real-time languages need a way to deal with time itself. To start with, a special variable, clock, should be available, containing the current time in ticks. One has to be careful about the unit that time is expressed in. The finer the resolution, the faster clock will overflow.

#### 1.4.6.3 Real-Time Communication

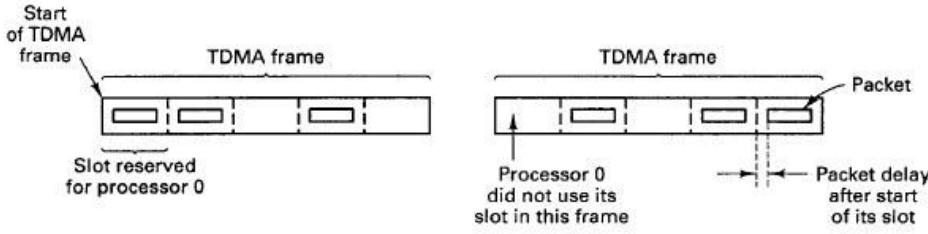
Communication in real-time distributed systems is different from communication in other distributed systems. While high performance is always welcome, predictability and determinism are the real keys to success. In this section we will look at some real-time communication issues, for both LANs and WANs. Finally, we will examine one example system in some detail to show how it differs from conventional (i.e. non-real-time) distributed systems.

Achieving predictability in a distributed system means that communication between processors must also be predictable. LAN protocols that are inherently stochastic, such as Ethernet, are unacceptable because they do not provide a known upper bound on transmission time. A machine wanting to send a packet on an Ethernet may collide with one or more other machines. All machines then wait a random time and then try again, but these transmissions may also collide, and so on. Consequently, it is not possible to give a worst-case bound on packet transmission in advance.

As a contrast to Ethernet, consider a token ring LAN. Whenever a processor has a packet to send, it waits for the circulating token to pass by, then it captures the token, sends its packet, and puts the token back on the ring so that the next machine downstream gets the opportunity to seize it. Assuming that each of the  $k$  machines on the ring is allowed to send at most one  $n$ -byte packet per token capture, it can be guaranteed that an urgent packet arriving anywhere in the system can always be transmitted within  $kn$  byte times. This is the kind of upper bound that a real-time distributed system needs.

Token rings can also handle traffic consisting of multiple priority classes. The goal here is to ensure that if a high-priority packet is waiting for transmission, it will be sent before any low-priority packets that its neighbors may have. For example, it is possible to add a reservation field to each packet, which can be increased by any processor as the packet goes by. When the packet has gone all the way around, the reservation field indicates the priority class of the next packet. When the current sender is finished transmitting, it regenerates a token bearing this priority class. Only processors with a pending packet of this class may capture it, and then only to send one packet. Of course, this scheme means that the upper bound of  $kn$  byte times now applies only to packets of the highest priority class.

An alternative to a token ring is the TDMA (*Time Division Multiple Access*) protocol shown in Fig. 1.34. Here traffic is organized in fixed-size frames, each of which contains n slots. Each slot is assigned to one processor, which may use it to transmit a packet when its time comes. In this way collisions are avoided, the delay is bounded, and each processor gets a guaranteed fraction of the bandwidth, depending on how many slots per frame it has been assigned.



**Fig. 1.34. TDMA (Time Division Multiple Access) frames.**

Real-time distributed systems operating over wide-area networks have the same need for predictability as those confined to a room or building. The communication in these systems is invariably connection oriented. Often, there is the ability to establish real-time connections between distant machines. When such a connection is established, the quality of service is negotiated in advance between the network users and the network provider. This quality may involve a guaranteed maximum delay, maximum jitter (variance of packet delivery times), minimum bandwidth, and other parameters. To make good on its guarantees, the network may have to reserve memory buffers, table entries, CPU cycles, link capacity, and other resources for this connection throughout its life-time. The user is likely to be charged for these resources, whether or not they are used, since they are not available to other connections.

A potential problem with wide-area real-time distributed systems is their relatively high packet loss rates. Standard protocols deal with packet loss by setting a timer when each packet is transmitted. If the timer goes off before the acknowledgement is received, the packet is sent again. In real-time systems, this kind of unbounded transmission delay is rarely acceptable. One easy solution is for the sender always to transmit each packet two (or more) times, preferably over independent connections if that option is available. Although this scheme wastes at least half the bandwidth, if one packet in, say, is lost, only one time in will both copies be lost. If a packet takes a millisecond, this works out to one lost packet every four months. With three transmissions, one packet is lost every 30,000 years. The net effect of multiple transmissions of every packet right from the start is a low and bounded delay virtually all the time.

**The Time-Triggered Protocol.** On account of the constraints on real-time distributed systems, their protocols are often quite unusual. In this section we will examine one such protocol, TTP (Time-Triggered Protocol) which is as different from the Ethernet protocol.

A node consists of at least one CPU, but often two or three work together to present the image of a single fault-tolerant, fail-silent node to the outside world. The nodes are connected by two reliable and independent TDMA broadcast networks. All packets are sent on both networks in parallel. The expected loss rate is one packet every 30 million years.

Clock synchronization is critical. Time is discrete, with clock ticks generally occurring every microsecond. TTP assumes that all the clocks are synchronized with a precision on the order of tens of microseconds. This precision is possible because the protocol itself provides continuous clock synchronization and has been designed to allow it to be done in hardware to extremely high precision.

All nodes are aware of the programs being run on all the other nodes. In particular, all nodes know when a packet is to be sent by another node and can detect its presence or absence easily. Since packets are assumed not to be lost (see above), the absence of a packet at a moment when one is expected means that the sending node has crashed.

For example, suppose that some exceptional event is detected and a packet is broadcast to tell everyone else about it. Node 6 is expected to make some computation and then broadcast a reply after 2 msec in slot 15 of the TDMA frame. If the message is not forthcoming in the expected slot, the other nodes assume that node 6 has gone down, and take whatever steps are necessary to recover from its failure. This tight bound and instant consensus eliminate the need for time-consuming agreement protocols and allow the system to be both fault tolerant and operate in real time.

Every node maintains the global state of the system. These states are required to be identical everywhere. It is a serious (and detectable) error if someone is out of step with the rest. The global state consists of three components:

1. The current mode.
2. The global time.
3. A bit map giving the current system membership.

The mode is defined by the application and has to do with which phase the system is in. For example, in a space application, the countdown, launch, flight, and landing might all be separate modes. Each mode has its own set of processes and the order in which they run, list of participating nodes, TDMA slot assignments, message names and formats, and legal successor modes.

The second field in the global state is the global time. Its granularity is application defined, but in any event must be coarse enough that all nodes agree on it. The third field keeps track of which nodes are up and which are down.

Unlike the OSI and Internet protocol suites, the TTP protocol consists of a single layer that handles end-to-end data transport, clock synchronization, and membership management.

The control field contains a bit used to initialize the system (more about which later), for changing the current mode and for acknowledging the packets sent by the preceding node (according to the current membership list). The purpose of this field is to let the previous node know that it is functioning correctly and its packets are getting onto the network as they should be. If an expected acknowledgement is lacking, all nodes mark the expected sender as down and expunge it from the membership bit maps in their current state. The rejected node is expected to go along with being excommunicated without protest.

The data field contains whatever data are required. The CRC field provides a checksum over not only the packet contents, but over the sender's global state as well. This means that if a sender has an incorrect global state, the CRC of any packets it sends will not agree with the values the receivers compute using their states. The next sender will not acknowledge the packet, and all nodes, including the one with the bad state, mark it as down in their membership bit maps.

Periodically, a packet with the initialization bit is broadcast. This packet also contains the current global state. Any node that is marked as not being a member, but which is supposed to be a member in this mode, can now join as a passive member. If a node is supposed to be a member, it has a TDMA slot assigned, so there is no problem of when to respond (in its own TDMA slot). Once its packet has been acknowledged, all the other nodes mark it as being active (operational) again.

A final interesting aspect of the protocol is the way it handles clock synchronization. Because each node knows the time when TDMA frames start and the position of its slot within the frame, it knows exactly when to begin its packet. This scheme avoids collisions. However, it also contains valuable timing information. If a packet begins n microseconds before or after it is supposed to, each other node can detect this tardiness and use it as an estimate of the skew between its clock and the sender's clock. By monitoring the starting position of every packet, a node might learn, for example, that every other node appears to be starting its transmissions 10 microseconds too late. In this case it can reasonably conclude that its own clock is actually 10 microseconds fast and make the necessary correction. By keeping a running average of the earliness or lateness of all other packets, each node can adjust its clock continuously to keep it in sync with the others without running any special clock management protocol.

### **1.4.7 Problems**

1. Imagine that a process is running remotely on a previously idle workstation, which, like all the workstations, is diskless. For each of the following system calls, tell whether it has to be forwarded back to the home machine:
  - (a) READ (get data from a file).
  - (b) (change the mode of the controlling terminal).
  - (c) GETPID (return the process id).
2. Can the model of triple modular redundancy described in the text handle Byzantine failures?
3. Does TMR generalize to five elements per group instead of three? If so, what properties does it have?

# PART II.

## ARCHITECTURES OF MODERN DISTRIBUTED SYSTEMS

## 2.1 Peer-To-Peer Architecture

### 2.1.1 A Brief History of P2P

Peer-to-peer (P2P) computing hit the front page headlines of technical Web sites in early 2000. It came on the scene as a new paradigm in network computing, a new technology for connecting people, and effectively utilizing untapped resources anywhere on the Internet and the Web.

P2P boasted of a network of equals, in which the traditional client/server partitioning of functionality and communication was replaced. The new paradigm of servants (a term formed from server + clients), partners in computing opportunity, was upon us. A number of factors unfolding in computing technology were driving this trend. P2P was changing the balance of power, and giving rise once again to the "power to the people" mantra of the 60s. Only this time, rather than protesting for personal individuality, the protest was for machine and silicon equality. A new generation of protesters took to the digital highways and soon caught the attention of the world.

The key factors changing the balance of power included the increased availability of inexpensive computing power, bandwidth, and storage. This, coupled with the explosion of content and subscribers on the Internet, started the next wave of the digital revolution.

The widespread adoption of Internet-based protocols enabled P2P champions to develop applications over a global network, a network never before seen in this light. Previous impediments to connectivity had been removed, and what shone through was a wealth of computing power, with access to a massive amount of content. Where the content resided was no longer an issue. It resided everywhere, within servers, within clients, within the fabric of the network itself. This liberal dissemination of data coupled with unlimited total access raised eyebrows.

Projections of digital asset ownership infringement and copyright abuse quickly became a reality. Users of Napster, the popular MP3 file sharing program, proliferated overnight. The exchange of MP3 files that month alone reached 2.7 billion, and control of content ownership was seen by the media as spinning out of control.

Proponents of P2P lined up to proclaim that freedom and liberty itself were under attack. We were witnessing firsthand as technology disrupted the status quo and brought into question the very laws that govern us. The battle being fought in the courtroom would determine the future of our society, and more importantly, P2P technology.

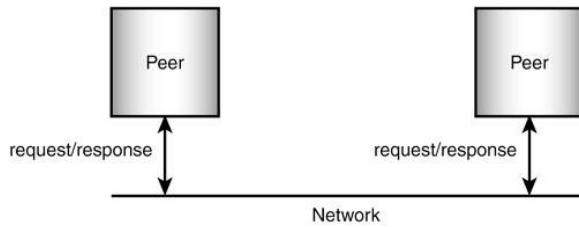
### 2.1.2 Definition and Motivation

P2P has a significant impact on the design and development of system architectures and commercial applications. It is having a profound impact on the Internet and the Web as we build the next generation of network-centric applications.

There is no universal agreement on what defines a P2P system. However, there are system characteristics that have surfaced, and a common nomenclature is being used to identify and describe P2P systems. It has become common practice to express the relationship between two computing entities as something-2-something. For example, B2B is defined as business-to-business; B2C is defined as business-to-consumer; and A2A is defined as application-to-application—the list continues to grow. These computing entities pose no problem of definition. Business is well understood, as is consumer and application; but what about peer?

The dictionary defines a peer as "a person who has equal standing with another or others, as in rank, class, or age." What stands out in this definition is the equality relationship: one person equal to another person in an important characteristic, such as rank, class, or age. Often these characteristics equate to the status or control an individual possesses within a community or society. So what you extrapolate from this definition is the notion of equality between computing entities. Fundamental to P2P is node equality, in which a node is defined as any processing entity that exists as a particular and discrete unit. See Figure 2.1.

If a person refers to characteristics such as rank, class, or age, what are the characteristics of a peer? What characteristics are we measuring to be equal? The simple answer is functionality. Peers are assumed to possess an equal capability in functions or services that they provide. This is contrary to traditional client/server models, in which the server possesses greater functionality and control than the client.



**Fig. 2.1. Peers represent liberated entities on a network of equals on which anyone can speak and listen.**

Napster was not pure P2P. All nodes did not possess equal processing capabilities. For instance, MP3 files were indexed on a central server; only the files remained on and were accessed from peer components. We will have more to say about mixed models such as this in a moment. What made Napster and other P2P applications interesting is the ease with which large networks of cooperating nodes could be assembled, and that these nodes lived on the edge of the network. The nodes were common PCs that dynamically assembled to form a distributed file system. The network was constantly changing. The nodes of the network were transient by nature. They would enter and leave the network at will. This is not the traditional approach to network organization, and is atypical of filesystems.

### 2.1.2.1 Peer-To-Peer Application

Consider the different types of applications a user with a desktop computer may expect to use today. In addition to a Web browser and "office" productivity suite, desktop users expect functions for the following:

- Managing and sharing information—Files, documents, photos, music, videos, and movies all want to be shared with business partners, friends, and colleagues. More advanced sharing enables one machine to act as a general task manager by collecting and aggregating results—for example, Google.com is an example of distributed task-managing system. Gnutella is an example of a personal P2P file-sharing system.
- Collaboration—Individual users find that address book, scheduler, chat and email software improves their productivity. Connecting the desktop productivity software together enables collaborative e-business communities to form for flexible, productive, and efficient working teams. For example, Java developers use OpenProjects.net to collaborate. On a broader scale, hundreds of thousands use instant messaging, which may be the most popular P2P application to date.
- Enterprise resource management—Coordinating workflow processes within an organization leverages the existing infrastructure of networked desktop computer systems. For example, Groove enables an aerospace manufacturer to post job order requests to partner companies and route the completed requests from one department to the next.
- Distributed computation—A natural extension of the Internet's philosophy of robustness through decentralization is to design peer-to-peer systems that send computing tasks to millions of servers, each one possibly also being a desktop computer.

P2P applications have emerged to satisfy the needs of users for all of these functions. As the benefits of P2P are better understood, it's likely that many more applications will be built using P2P technologies.

## Motivation to Adopt P2P

Three primary business and technology issues are driving the adoption of P2P:

- Decentralization—Businesses realize greater efficiency and profits by attaining a flexible state. Consequently, business leaders have been decentralizing for decades: from mainframes, to a client/server model, to Internet computing, and now to P2P. The trend is undeniably decentralized and distributed.
- Cost and efficiency—Hardware and software will continue to be inexpensive and powerful. New systems that increase the efficiency or utilization of hardware or software present a compelling case for making the investment. P2P additionally has the capability to exploit resources that in the past went unrecognized.
- Pervasive computing—Imagine information systems everywhere: computer chips in clothing, appliances, automobiles, devices, anything you can think of. Not only are they everywhere, but they are connected. The market for network-connected devices continues to grow, and P2P systems are being designed to support the device market.

As enterprises advance their e-commerce efforts, they are increasingly recognizing the need to couple transaction flows and communication flows. Similarly, they are also recognizing the natural tendency for co-workers and their external counterparts to establish "communities" in order to perform both routine and special-purpose tasks. Growth in e-commerce should fuel more demand for collaborative commerce. P2P technology is a natural fit for this growing trend to establish distributed special-purpose communities.

## **2.1.3 Functionality**

### **2.1.3.1 P2P Architectural Details**

What makes P2P applications interesting is the ease with which large networks of cooperating nodes can be assembled, and that these nodes live on the edge of the network. The nodes are common PCs that dynamically assemble to form a distributed file system. The network is constantly changing—many nodes are behind firewalls and one-way Network Address Translation (NAT) routers; the computers may be turned off at night, and they enter and leave the network at will. This is antithetical to the network organization and typical filesystems found on business networks.

In a client/server model, the server controls and manages the relationship clients have with resources, such as databases, files, networks, and other clients. The server functions as a higher-level citizen within the computing community. It is given special privileges and functionality to control its subjects.

Node equality has a dramatic impact on the way we architect and build systems. What has been solved by traditional hierarchical systems is now unraveled, and up for debate and re-evaluation in the peer-to-peer world. How do we identify and locate entities? Who controls access to resources? Although these questions are difficult to answer in any environment, they are at least well understood. In P2P systems, this is not the case. This is the new frontier of computing. The rules have not been defined, and the opportunity still exists to engage in design and development on the edge. P2P may be the first wave of delivering post Web-browser Internet content.

Interestingly, the definition of the verb tense of the word peer is "to look intently, searchingly, or with difficulty." This definition can be applied to the actions required by P2P nodes when you remove hierarchical relationships. How do you search for peers and form groups of cooperating entities? The first generation of P2P applications grappled with the problems inherent in this question. As a result, they exposed the strengths and weaknesses of early P2P systems. One can learn a lot about P2P by studying early systems like Napster, Gnutella, and Freenet. Applications such as these revealed common characteristics of P2P systems. P2P systems have a dynamic element that enables them to form or discover groups and communities. Early systems used this primarily for searching, or to solve a common problem.

P2P systems require virtual namespaces to augment current addressing technology. A virtual namespace provides a method for persistent identification, which would otherwise not be possible. For the moment, think of this as your email address that uniquely identifies you, regardless of what computer you use to access your mail.

Peers in a P2P system are considered equals in terms of functional capabilities. Equality means you no longer need an intermediary to help you participate in a network. The connection to Internet is sufficient to get involved. Peers can appear anywhere on the network. They can be your PC, or the Palm Pilot that you hold in your hand. If you can connect it to the network, you can "peer" it.

Peers need not be permanent; they have a transient capability to appear and disappear on the network. Intermittent connectivity in many P2P systems is the norm rather than the exception. Early P2P systems were comprised of dial-up users who established connectivity, joined the network, and then disconnected and left the network. P2P systems had to account for this type of membership.

Peers have a wide array of processing, bandwidth, and storage capabilities. While they are all equal, some are more equal than others. A laptop computer can connect to the Internet through a dial-up connection and become a peer. A Sun Enterprise 10000 with fiber optic pipes can also become a peer on the same network. Functionally, in the P2P system they are equal. However, their performance capabilities are quite different. P2P is changing the way we build systems that exploit the global network.

### **2.1.3.2 How P2P Forms Dynamic Networks**

Dynamic networks are fundamental to P2P systems. The Internet is a dynamic network with a number of static properties. For example, each machine that connects to the Internet is assigned a unique IP address.

IPv4, the predominant protocol today, uses 32-bit addresses. Values are represented in decimal notation separated by dots; for example, 172.16.1.2. This configuration limits the possible addresses that are available. The proliferation of user machines and devices requiring IP addresses has gone beyond the original creators' vision. We are running out of addresses. The IPv6 protocol has been defined to extend the range of possible addresses, and to be backward compatible with IPv4. IPv6 uses 128-bit addresses. Values are represented as hexadecimal numbers separated by colons; for example, FEDC:B978:7654:3210:F93A:8767:54C3:6543. IPv6 support  $10^{12}$  (1 trillion) machines and  $10^9$  (1 billion) individual networks.

Because humans remember names more easily than numbers, the Internet provides a way for us to use names to identify machines. The Domain Name Service provides the mechanism that helps users identify or map a machine name to an IP address. As a result, we can use <http://java.sun.com> rather than <http://192.18.97.71/>.

Although you can use IP and DNS to identify and find certain machines on the network, there still exist challenges for P2P systems. The limited number of IP addresses available using IPv4 has resulted in additional identification mechanisms. NAT makes it possible to assign a pool of reserved IP addresses to machines on a local network. When connecting to the Internet, the machines share a "public" IP address. Because the reserved pool has been set aside for use in private networks, they will never appear as public addresses. Consequently, they can be reused. Although these mechanisms do wonders to conserve addresses, they make discovering real machine addresses difficult, especially in dynamic environments. The next-generation Internet, which will use IPv6, is designed to address this problem, but it's also likely years in the future. In the meantime, dynamic IP assignment on the Internet is still common, and creates an inherent identification problem.

How do you recognize a peer that no longer has the same identity? Peer-to-peer networks must be able to uniquely identify peers and resources that are available. As a result, P2P systems have had to define their own naming schemes independent of IP addresses or DNS. They have had to create virtual namespaces, enabling users to have persistent identities on their systems.

Rather than being predefined or preconfigured such as in DNS, the nodes within the network "find" or "discover" each other using IP and DNS as a navigational aid to build a dynamic or virtual network. Dynamic network formation is typical of P2P networks.

### **2.1.3.3 Discovery**

How peers and resources of the P2P system are discovered has generated a substantial amount of press and dialogue. To date, it has been the elusive measure of success for peer-to-peer systems.

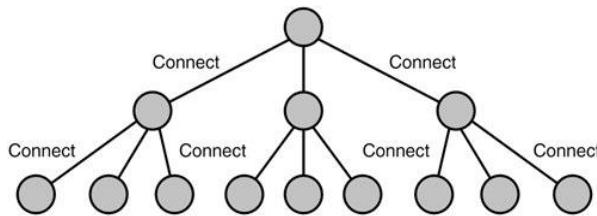
One can think of discovery on two levels. First, the discovery process is associated with finding a peer. In this case, a peer refers to a computing entity that is capable of understanding the protocol of the messages being exchanged. It is an entity that "speaks" the same language—it understands the semantics of the dialogue. Peer discovery is required to find a service or help divide and conquer many problems associated with information processing. If we didn't understand what we were exchanging, we couldn't progress beyond digital babble.

The second level of discovery is associated with finding resources of interest. The early P2P applications dealt with file sharing and searching. In contrast to popular search engines, P2P applications define new techniques to discover files and information on the Internet.

The massive amount of information available on the Internet and its exponential growth is outpacing traditional information indexing techniques. In addition, the delay between content availability and content discovery continues to grow despite parallelism in popular search engines. P2P resource discovery provides a more real-time solution to information searching. However, the discovery techniques and protocols required have come at a price.

The Gnutella story has been well documented. A popular file sharing and search program, Gnutella uses an unconventional broadcast mechanism to discover peers, as illustrated in Figure 2.2. The broadcast technique

grows exponentially—the more users, the more broadcasts. When the size of the user base grew too quickly, the system came crashing to a halt, flooding networks with Gnutella requests. The success of the software highlighted the limitations of its discovery architecture.



**Fig. 2.2. Gnutella discovery quickly ran into the "broadcast storm" problem once the network grew beyond initial expectations.**

An effective discovery mechanism is critical to the successful design and deployment of a peer-based network. To be effective, a discovery mechanism must be efficient in different execution environments. It should be efficient in discovering peers and resources regardless of the size of the network. It should also be resilient enough to ward off attacks and security breaches that would otherwise jeopardize the viability of the technology.

Centralized methods of discovery often break down when applied to large peer-based networks. They often fail to scale or present single points of failure in the architecture.

There are a number of decentralized discovery methods in use that use a variety of designs and architectures. All of these methods have various strengths that make them attractive for certain circumstances. However, they all have tradeoffs in large peer-based networks.

#### 2.1.3.4 Broadcast

**Simple broadcast** sends a request to every participant within the network segment or radius. When used for discovery, it can reach a large number of potential peers or find a large number of resources. The drawback to this approach is that as the user base grows linearly, the number of requests grows exponentially.

This approach can result in huge bandwidth requirements. At some point, the network will be saturated with requests and trigger timeouts and re-transmissions, which just aggravates the already dire situation. There are also security and denial-of-service implications. A malicious peer can start flooding the network with a number of requests disproportionate to the true size of the user base. This can interrupt the network and reduce its effectiveness. Also, simple broadcast is only viable in small networks.

A variation on simple broadcast is **selective broadcast**. Instead of sending a request to every peer on the network, peers are selected based on heuristics such as quality of service, content availability, or trust relationships. However, this type of broadcast requires that you maintain historical information on peer interactions.

Discovery requests are sent to selected peers, and the response is evaluated against the criteria that you have defined for peer connections. For instance, you might only send discovery requests to peers that support a certain minimum bandwidth requirement. Or you might send requests for resources only to peers likely to have that content. Of course, the more you need to know about the participants, the less dynamic the system can become. This can quickly eliminate the benefits of P2P if fixed and static relationships are not mitigated.

Security is still a concern with selective broadcast. It is important that each one of the peers be reputable for this operation to be effective.

Like selective broadcast, **adaptive broadcast** tries to minimize network utilization while maximizing connectivity to the network. Selection criteria can be augmented with knowledge of your computing environment. For instance, you can set the amount of memory or bandwidth you will consume during discovery operations. You can limit the growth of discovery and searching by predefining a resource tolerance level that if exceeded will begin to curtail the process. This will ensure that excessive resources are not being consumed because of a malfunctioning element, a misguided peer, or a malicious attack. Adaptive broadcast requires monitoring resources, such as peer identity, message queue size, port usage, and message size and frequency. Adaptive broadcast can reduce the threat of some security breaches, but not all.

### **2.1.3.5 Resource Indexing**

Finding resources is closely tied to finding peers. However, the difference is that peers have intelligence; they are processes capable of engaging in digital conversations through a programming interface. A resource is much more static, and only requires identity. Discovering resources can be done using centralized and decentralized indexing. Centralized indexes provide good performance, at a cost. The bandwidth and hardware requirements of large peer networks can be expensive. Centralized indexes hit the scalability wall at some point, regardless of the amount of software and hardware provided. Decentralized index systems attempt to overcome the scalability limitations of centralized systems. To improve performance, every document or file stored within the system is given a unique ID. This ID is used to identify and locate a resource. IDs easily map to resources. This approach is used by FreeNet. The drawback to this approach is that searches have to be exact. Every resource has a unique identifier.

Another problem with decentralized indexing systems is keeping cached information consistent. Indexes can quickly become out of sync. Peer networks are much more volatile, in terms of peers joining and leaving the network, as well as the resources contained within the index. The overhead in keeping everything up to date and efficiently distributed is a major detriment to scalability.

Because peer networks are so volatile, knowing when a peer is online is required to build efficient and user-centric distributed systems. P2P systems use the term presence, and define it as the ability to tell when a peer or resource is online. The degree to which this situation affects your environment is application-dependent; however, you must understand the implications.

### **2.1.3.6 Node Autonomy**

P2P systems are highly decentralized and distributed. The benefits of distribution are well-known. You generally distribute processing when you need to scale your systems to support increased demand for resources. You also distribute for geographic reasons, to move resources and processes closer to their access point. Other reasons to distribute are to provide better fault resistance and network resilience, and to enable the sharing of resources and promote collaboration.

Decentralization gives rise to node autonomy, and in a peer-to-peer system, peers are highly autonomous. Peers are independent and self-governing. As mentioned, in a client/server model, the server controls and manages the relationship clients have with resources, such as databases, files, networks and other clients. This has many advantages in the operation, administration, and management of a computing environment. One of the advantages of centralization is central administration and monitoring. Knowing where resources are and how they are behaving is a tremendous advantage. Resources can be secured and administered from a central location. Functionality can be deployed to complement the physical structure of the network topology. For instance, servers can act as gatekeepers to sensitive technology assets.

With decentralization comes a number of significant challenges: The management of the network is much more difficult. In a distributed environment, failures are not always detected immediately. Worse yet, partial failures allow for results and side effects that networks and applications are not prepared to deal with. Response time and latency issues introduced as a result of remote communication can be unpredictable. The network can have good days and bad days. Peer-to-peer interaction can become unstable as error paths and timeouts get triggered excessively. Synchronization often strains available bandwidth.

Any solution that is based on distribution should be able to eliminate or mitigate these issues. P2P systems are built under the assumption that services are distributed over a network, and that the network is unreliable. How P2P systems cope with unreliable networks differentiates one system from another.

### **2.1.3.7 Peer of Equals**

Peers in a peer-to-peer system have the capability to provide services and consume services. There is no separation of client versus server roles. Any peer is capable of providing a service or finding a peer that can provide the service requested. A peer can be considered a client when it is requesting a service, and can be considered a server when it is providing a service.

Peers are often used in systems that require a high level of parallelism. Parallelism is not new to computing. In fact, much of what we do in computing is done in parallel. Multiprocessor machines and operating systems rely on the capability to execute tasks in parallel. Threads of control enable us to partition a process into separate tasks. However, to date, parallelism has not been the norm in application development. While applications are designed to be multithreaded, this generally has involved controlling different tasks required of a process, such as reading from a slow device or waiting for a network response. We have not defined many applications that run the same tasks in parallel, such as searching a large database, or filtering large amounts of information concurrently.

Parallelism can provide us with a divide-and-conquer approach to many repetitive tasks. The SETI@Home project demonstrated that personal computers could be harnessed together across the Internet to provide extraordinary computing power. The Search for Extraterrestrial Intelligence project examines radio signals received from outer space in an attempt to detect intelligent life. It takes a vast amount of computing power to analyze the amount of data that is captured and the computations involved. People volunteered for the project by downloading a screensaver from the SETI Web site. The screensaver was capable of requesting work units that were designed to segment the mass amount of radio signals received. In the first week after the project launched, more than 200,000 people downloaded and ran the software. This number grew to more than 2,400,000 by the end of 2000. The processing power available as a result of this network outpaced the the fastest supercomputer built at the time. P2P systems are designed to meet this growing trend in divide-and-conquer strategies.

The SETI project is typical of system architectures that require a certain degree of centralization or coordination. Networks can be classified by their topology, which is the basic arrangement of nodes on the network. Different types of network configurations exist for network designers to choose from.

A decentralized topology is often augmented with a centralized component, which creates a mixed model, or hybrid architecture. With Napster, it's the centralized file index component that's capable of identifying and locating files. With SETI, it's the centralized task dispatcher that allocates work units.

### **2.1.3.8 Supporting Mixed Models**

Many P2P technologies are now adopting a network-based computing style that supports a mixed model. The predominant decentralized model is augmented with centralized control nodes at key points. The architectures define central control points for improving important performance characteristics of P2P systems, such as discovery and content management. Hybrid architectures can also enhance system reliability and improve the fault tolerance of systems.

### **2.1.4 The Future Includes Web Services**

The vision behind Web services is very compatible to P2P technology. Web services enable developers to build loosely coupled, self-describing, highly scalable systems that provide interoperability between software on different platforms.

Software developers working on traditional client/server and Web applications are challenged with the same issues: scalability, interoperability, performance, and maintenance. P2P technology is beginning to look very attractive as a tool for designing high performance, scalable, server-based systems.

P2P applications provide an integration model for Web services that works alongside the Simple Object Access Protocol (SOAP). Even with the use of SOAP, Web applications are typically designed to include a Web server and a browser client. Web applications usually result in centralized data centers and spread-out populations of clients. Centralized repositories have their place, but so do decentralized implementations, where the services provided at the edge of the network power an application. Imagine a stock trading application in which the trades are handled by a centralized server using SOAP, and the stock charting and history functions come from a network of information sources using P2P.

The software industry is headed toward larger interoperability, and P2P has already found a viable place. Device management, service monitoring, and resource management applications are being developed to support and complement current network management protocols, and e-commerce applications are being defined using P2P as a technology enabler, including the following:

- Online auctions
- Pricing and payment models
- Trading hubs and e-commerce communities
- Customer care
- Services for mobile users
- Service personalization

Although this chapter presents a good starting point, there are issues that you will need to understand to determine the applicability of P2P in your system selection and development.

### **2.1.5 P2P Application Types**

It might have appeared that the definition of P2P is problematic—well, it is! P2P can be so broad in scope and definition that getting one's arms around P2P can be an enormous task. P2P came about as an answer to user needs for Internet-enabled application software. The Internet is always evolving and offering up new technologies, techniques, and user behavior every day. P2P is evolving along with the Internet, so fixed definitions do not usually last long.

Each new advance in Internet technology can either help Java developers working on Internet applications, or become a huge headache. For example, there was a time when Network Address Translation (NAT) routers were banned from networks. As you will see later in this chapter, they are now used widely, and a Java developer building a P2P application needs a solution to the unique one-way routing provided by a NAT router.

An easy way to get your arms around a P2P definition is to look at the functions delivered by the most notable P2P applications, including the following:

- Instant messaging
- Managing and sharing information
- Collaboration

What started out as simple file sharing, such as exchanging music files, has grown to include a wide array of applications and services. These are grouped under the umbrella term distributed P2P services. These include network and infrastructure software to enable

- Distributed processing (grid computing)
- Distributed storage
- Distributed network services

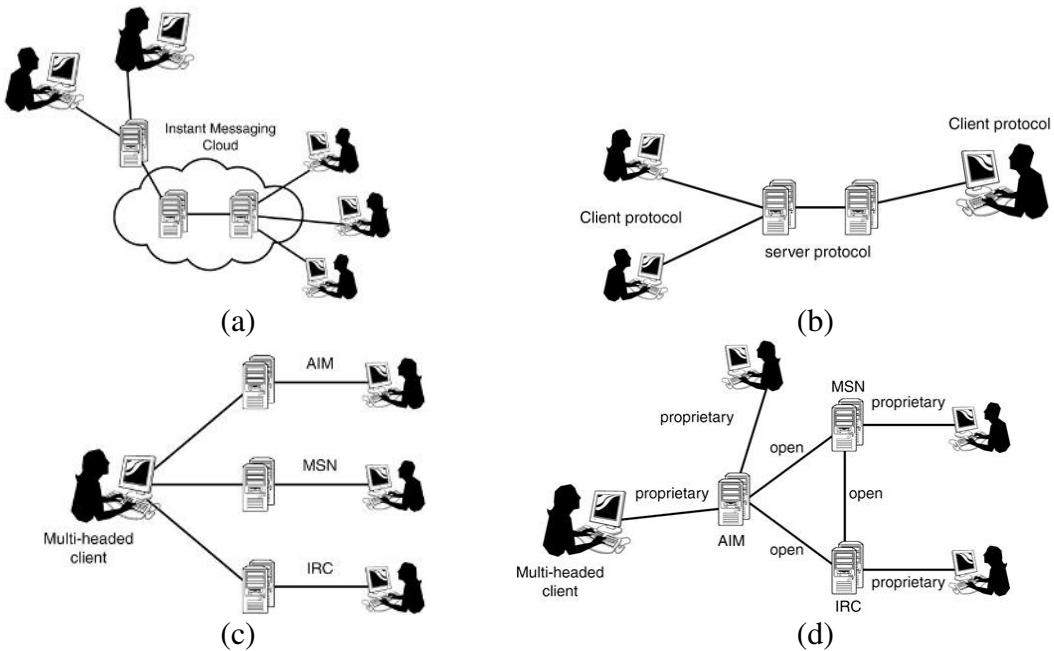
Although many of these applications began as ways to distribute stolen copyrighted music and video files, P2P has reached a level of maturity that is no longer confined to personal, casual use, but rather to build e-market hubs, corporate infrastructure, and Internet-enabled applications. In addition, single-function P2P applications are giving way to multifunction service-based architectures. For instance, it is common to aggregate instant messaging, file sharing, and content management to build distributed collaborative P2P applications.

#### **2.1.5.1 Instant Messaging**

Although Web publishing and browsing is the killer application for the Internet, instant messaging is the killer application for P2P. Instant messaging (IM) enables online users to communicate immediately and in real-time, one-to-one or in a group. It has become popular on the Internet among young adults, and is gaining popularity in business settings, too. For example, even IBM's Lotus group offers an IM product for business use. IM has gained recognition as a useful application, and most major Internet players offer IM services—AOL, Microsoft, and Yahoo! all offer IM functionality. New players are promoting niche products for e-commerce and supply chain management. There has been some market consolidation as lesser offerings have been eliminated, but there is still tremendous growth and opportunity.

With IM, users activate a special piece of client software that communicates with a central server and registers the user as being online. This user registration is mapped to an identity, such as a nickname or screen name. The user is then able to invite others to a conversation, or can be invited. IM servers communicate using an IM server protocol that enables messages to be relayed across the Internet. This interconnected IM network forms an IM

cloud, or backbone, as shown in Figure 2.3(a). Most servers support a proprietary protocol, which has made it difficult for IM users to communicate across multiple IM systems.



**Figure 2.3 (a)** Traditional instant messaging systems enable users to exchange messages through a proprietary IM cloud. **(b)** Instant messaging systems define a client and a server protocol. **(c)** Multi-headed clients enable a user to communicate with multiple IM systems. **(d)** Interoperability is achieved at the service level.

Unlike email, in which a message is stored and delivered once the user has connected to an email server, IM systems provide immediate end user delivery. If the user is not available, the message can be saved until the user comes online, or it simply may be discarded. To avoid this uncertainty in delivery, IM systems provide a "buddy list" or roster that provides a mechanism to identify a user and determine the user's online status: for example, online, offline, or unavailable.

If the user is online, you can send text messages that are immediately delivered to the user. This promotes a two-way conversational style of communication with minimal delay. Tight integration between clients and servers enables instant messaging services to provide varying levels of security, online status, and reliable messaging, as seen in Figure 2.3(b). The client protocol defines the message structure necessary to communicate short text messages. The server protocol defines the higher-level services, such as routing, presence, and security.

Applications of presence (online status) and instant messaging currently use independent, nonstandard, and non-interoperable protocols developed by various vendors.

Because instant messaging has become so popular, it is not surprising that there is an abundance of IM providers. Today, many P2P applications are including an instant messaging component. It is envisioned that as businesses link their manufacturing, distribution, and sales processes, more reliance on IM features will be the result. Many Web sites already use IM as a key component in their customer relationship management (CRM) strategy.

**IRC.** Internet Relay Chat ([www irc org](http://www irc org)), is often cited as the original chat medium on the Internet. It is the forefather to many of the IM protocols that have been developed. The IRC protocol was designed for use with text-based conferencing. The IRC protocol is based on the client-server model. A server forms a central point for clients (or other servers) to connect to, and performs the required message delivery, multiplexing, and IM functions. Over the years, IRC has been expanded and changed to resolve scalability problems. For example, while the original IRC protocol required a single central server, today's IRC enables federations of interconnected servers to pass IM messages. OpenProjects.net ([www openprojects net](http://www openprojects net)) provides a backbone of IRC servers that developers and users working on open-source projects use. IRC continues to evolve.

**Jabber** ([www jabber org](http://www jabber org)) is an open source instant messaging platform being developed by the open source community. One of the features that distinguishes the Jabber system from existing instant messaging services is its open XML protocol. The Jabber protocol has been submitted as an IETF Request For Comments (RFC). Jabber is

attempting to build the interoperable protocol that all IM vendors will support. This would enable the interoperability that is envisioned by the IM community. In the meantime, the Jabber architecture is built on pluggable transport modules that communicate with specific IM systems. The idea is that you use the Jabber XML protocol (XMPP) from the client to the Jabber server, and the server loads an IM-specific transport module to interoperate with the proprietary IM system. The Jabber architecture resembles email. A Jabber client is connected to a Jabber server. Like an email server, the Jabber server is responsible for the delivery and receipt of the client's messages. However, Jabber servers will attempt to deliver the messages immediately, thereby supporting instant messaging and conversational capabilities. The Jabber server will queue messages when a peer is unavailable or offline. The peer-to-peer comparisons of Jabber are more appropriately realized with the relationship between Jabber servers. Every Jabber server is a peer to every other Jabber server. Jabber servers use a number of mechanisms to improve the integrity and security of the system. For instance, hostname dialback independently contacts the sending server to validate incoming data to prevent spoofing.

### 2.1.5.2 Managing and Sharing Information

The next category of P2P services distributes the management and sharing of computer resources to a group of peers across the network. A number of subcategories are included:

- File sharing
- Resource sharing
- Distributed search engines

File sharing applications such as Gnutella and Freenet form ad hoc P2P communities. These applications share files without requiring centralized coordination or control. Resource sharing applications are a form of distributed computing that use of the cumulative power of dynamically networked peers to tackle tasks previously possible only on supercomputers. For example, the SETI project is one example of resource sharing.

Finally, distributed search engines are a P2P technology that address the problems inherent in the large size of the information space. Distributed search engines push the computing functions needed to build an index of search results toward the edge of the network where peers live. Distributed search engines use a divide-and-conquer strategy to locate information and perform these searches in real time.

Like instant messaging, the popularity of file and resource sharing has created a large market for products.

**Gnutella** was a popular file sharing and searching protocol that operates with edge devices in a decentralized environment. In the Gnutella network, each peer acts as a point of rendezvous and a router, using TCP for message transport, and HTTP for file transfer. Searches on the network are propagated by a peer to all its known peer neighbors who propagate the query to other peers. Content on the Gnutella network is advertised through an IP address, a port number, an index number identifying the file on the host peer, and file details such as name and size. The Gnutella client LimeWire, (<http://www.limewire.com/>) is a software package that enables individuals to search for and share computer files over the Internet. LimeWire is compatible with the Gnutella file sharing protocol, and can connect with any peer running Gnutella-compatible software. At startup, the LimeWire client connects via the Internet to the LimeWire gateway. The LimeWire gateway is an intelligent Gnutella router. LimeWire is written in Java, and run on any machine with an Internet connection and the capability to run Java.

**NextPage** bridges the consumer-oriented Internet architectures with corporate intranets and extranets. The company is adopting a P2P strategy with its server-oriented architecture. NextPage's NXT-3 Content Networking platform enables users to "manage, access, and exchange content across distributed servers on intranets and via the Internet." The platform indexes and connects content across organizational boundaries, allowing you to search and locate content from multiple locations without having to know where it physically resides. NextPage offers an extensive array of search functions including keyword, Boolean, phrase, ranked, wildcard, and so on. For more information, go to [www.nextpage.com](http://www.nextpage.com).

### 2.1.5.3 Collaboration

Individuals and businesses no longer are bound by specific tools within a community or geographic region. Common interests and objectives are the driving force. The tools to support the formation and integration of decentralized communication and ad hoc group formation already exist. For example, the popularity of AOL Instant Messenger and ICQ have demonstrated that real-time communication is a viable and often preferred way

to communicate. Products such as Napster and FreeNet have proven that decentralized file sharing is possible. Collaboration is the next step. Collaboration increases productivity, and enables teams in different geographic areas to work together. As with file sharing, it can decrease network traffic by reducing email, and decreases server storage requirements by using edge devices to store projects and information locally.

Shared spaces form the backbone to P2P collaborative applications:

- Rendezvous points—Shared spaces enable peers to identify a common network accessible meeting place.
- Identity and presence services—Shared spaces become the common point for searching, retrieving, and updating identity and online status.
- Group membership—Shared spaces form the basis for defining a group or community of peers connected by a common interest or goal. Group membership in a shared space is controlled by the group, rather than by a central administrator.

**Groove** is a rethinking of the Lotus Notes architecture. Notes slightly gained popularity before the Web became very popular in the 1990s. Groove attempts to overcome Notes' Internet limitations by delivering a P2P platform to developing and deploying secure enterprise applications. At first, Groove Networks positioned Groove as a P2P collaborative platform. Recent moves to align itself with Microsoft have shown some promising uses of P2P technology in Windows environments. Although nothing has been publicly discussed at the time of this book's writing, Groove is definitely worth keeping an eye on. Already Groove has found a home in enterprises, government, small businesses, and individuals. It offers instant collaboration, shared spaces, Web connectivity, and a host of add-on applications. Developers can integrate Groove into their existing systems. Groove and other competing products not only provide the capability to access data on traditional corporate networks, but also in nontraditional devices such as PDAs and a range of handheld devices.

#### 2.1.5.4 Distributed Services

Corporations increasingly see e-commerce, or collaborative commerce as it is often called, as a strategic tool. Corporations list the top goals of collaborative commerce as

- Reducing the cost of operations
- Gaining competitive advantage
- Increasing customer loyalty and retention

Collaborative commerce is being built on distributed services. Initially the focus is on Web services. Complementary P2P technologies might turn out to be the enabler that Web services need to fulfill corporate expectations.

Distributed P2P services for collaborative commerce fall in three categories; distributed data services, distributed computing, and distributed network services.

**Distributed data services** move data closer to usage using multiple nodes and sophisticated routing algorithms. Peer-based storage has distinct advantages:

- Edge devices are plentiful and under-used.
- Users are more mobile, and must access information from multiple locations.
- Centralization has proven costly and prohibitive beyond a certain level of scalability.
- Mobile users are demanding faster access to content.
- Service providers are searching for cheaper solutions.

There are a number of complexities to implementing successful distributed content networks. Distributed content networks require intelligent caching over a widely dispersed cluster of nodes. Some distributed content networks use predictive seeding to preconfigure the location of data based on usage patterns and known heuristics. Multisourcing permits a content network to map multiple communication paths to a cluster and/or data store. This is often accomplished through intelligent routing. Security (encryption) is required to ensure integrity of data in transit. Presence and bandwidth matching guarantees that nodes are available, and that they map performance capabilities appropriately. Content delivery networks automatically detect which peer servers are available on the network to serve the file segments with the greatest overall efficiency.

**Distributed computing** (also referred to as grid computing) attempts to use the idle processing cycles of the PCs on your network. It makes these commodity devices available to work on computationally intensive problems that

would otherwise require a supercomputer or workstation/server cluster to solve. There are usually three fundamental components to the architecture:

- The Network Manager manages client resources, and controls which applications are run on the client machines.
- The Job Manager permits application users to submit work and monitors the progress of this work, and retrieves results.
- The Client manages the running of applications on a client machine.

Distributed computing should continue to gain in popularity as more sophisticated problems surface, such as in genetics and bioinformatics.

The attractiveness of **distributed network services** lies in their capability to localize traffic, lowering bandwidth expenditures and improving response times. By serving and fulfilling as many requests for data from devices on the LAN, enterprises and ISPs can cut costs and improve performance dramatically. Bandwidth management technologies give network administrators at Internet service providers or corporate enterprises the capability to set and enforce policies to control network traffic, ensuring that networks deliver predictable performance for mission-critical applications. Bandwidth management systems can prioritize mission-critical traffic, as well as guarantee minimum bandwidth for the most critical, revenue-generating traffic (for example, voice, transaction-based applications). Webcasting continues to grow, and bandwidth demands continue to increase. Distributed P2P network services automatically scale to use the available bandwidth and computer resources of new participants who request the stream. This is a more economical model for usage in high bandwidth-intensive applications that require infrequent activation. Load balancing/traffic management products process network traffic streams, switching and otherwise responding to incoming requests by directing these requests to specific server clusters based on a set of predefined rules. These products usually have the capability to test the servers they are connected to for correct operation, and reroute data traffic around a server should it fail. Most of these devices also have the capacity to recognize the requester or the data being requested, and prioritize the request or the response accordingly. The capability of the load balancer/traffic manager to consider the source of the message or the relative load on each of several replicated servers and then direct the message to the most appropriate server increases efficiency and uptime.

## 2.1.6 JXTA

JXTA made two significant contributions to P2P technology. First, JXTA provided huge validation for building all future P2P technology with XML. JXTA protocols use self-describing XML definitions to move messages and manage the environment. Second, by providing a flexible and quality reference implementation, JXTA takes the Java development communities' focus away from basic P2P coding and on to solving P2P problems. P2P applications have matured, and are looking to increase interoperability among peers by defining standard protocols. P2P technologies will become more integrated into the network infrastructure so that more applications can utilize peer-computing services.

JXTA defines three layers. The bottom layer addresses the communication and routing and P2P connection management. The middle layer handles higher-level concepts, such as indexing, searching, and file sharing. The top layer provides protocols that the applications use to manage the middle-layer services and lower-layer "plumbing" to build full-featured P2P applications. The top layer is where typical applications, such as instant messaging, network services, and P2P collaboration environments are defined. In JXTA, all protocols are defined as XML messages sent between two peers. JXTA messages define the protocols used to discover and connect peers and to access network services offered by peers and peer groups.

Each JXTA message has a standard format, and may include optional data fields. So, JXTA standardizes the messages exchanged between peers by defining standard XML data streams used to invoke common functions or features of P2P services.

Messages are sent between logical destinations (endpoints) identified by a URI. The transport must be capable of sending and receiving datagram-style messages. Endpoints are mapped into physical addresses by the messaging layer at runtime.

## 2.2 Grids

Before 1990, the world wide Internet network was almost entirely unknown outside the universities and the corporate research departments. The common way of accessing the Internet was via command line interfaces such as telnet, ftp, or popular Unix mail user agents like elm, mush, pine, or rmail. The usual access to information was based on peer-to-peer email message exchange which made the every day information flow slow, unreliable, and tedious. The advent of the World Wide Web has revolutionized the information flow through the Internet from obsolete message passing to world wide Web page publication. Since then, the Internet has exploded to become an ubiquitous global infrastructure for publishing and exchange of (free) digital information.

Despite its global success and acceptance as a standard mean for publishing and exchange of digital information, the World Wide Web technology does not enable ubiquitous access to the billions of (potentially idle) computers simultaneously connected to the Internet providing peta-flops of estimated aggregate computational power. Remote access to computational power is highly demanded by applications that simulate complex scientific and engineering problems, like medical simulations, industrial equipment control, stock portfolio management, weather forecasting, earthquake simulations, flood management, and so on.

Grid computing represents the next evolution step after cluster computing in aggregating cheap and widely available computing power required by high performance scientific applications. The scope of Grid computing is to aggregate and provide coordinated use of large sets of heterogeneous distributed resources, ranging from sequential and parallel computers, to storage systems, software, and data, all connected through the Internet-based wide area (high performance) network. In contrast to dedicated clusters characterized by close (local area) proximity of their computing nodes and often homogeneous hardware and software infrastructures, Grids are characterized by the distant proximity of large numbers of aggregated sites, inherently heterogeneous in terms of hardware, operating systems, and software.

There are currently two recognized architectural approaches for building scalable Grid infrastructures:

1. **Peer-to-peer architectures** are an aggregation of equivalent programs called *peers* situated at the edges of the Internet that provide functionality and share part of their own hardware resources with each other (e.g. processing power, storage capacity, network link bandwidth, printers) through network contention without passing through intermediate entities. The strength of peer-to-peer architectures is the high degree of *scalability* and *fault tolerance*.
2. **Service-oriented architectures** are based on the aggregation of portable and reusable programs called services that can be accessed by remote clients over the network in a platform and language independent manner. A service is *a self-contained entity program accessible through a well-defined protocol and using a well-defined platform and language independent interface that does not depend on the context or the state of other services*.

### 2.2.1 Computational Grids

Nowadays, the common policy of accessing high-end computational resources is through manual remote ssh logins on behalf of individual user accounts. Similar to the World Wide Web that revolutionized the information access, **computational Grids** are aiming to define an infrastructure that provides dependable, consistent, pervasive, and inexpensive access to the world wide computational capabilities of the Internet. In this context, computational Grids raise a new class of important scientific research opportunities and challenges regarding, e.g.:

- secure resource sharing among dynamic collections of individuals and institutions forming so called *Virtual Organizations*;
- solving large scale problems for which appropriate local resources are not available;
- improving the performance of applications by increasing the parallelism through concurrent use of distributed computational resources;
- coarse-grain composition of large-scale applications from off-the-shelf existing software services or application components;
- exploiting (or stealing) unused processor cycles from idle workstations (e.g. desktop, network) to increase the overall computational power;
- appropriate distribution and replication of large data files near to places where subsequent computations will likely take place;
- incorporation of semantic Web technologies.

In the past years, the interest in computational Grids has increasingly grown in the scientific community as a mean of enabling application developers to aggregate resources scattered around the globe for solving large scale scientific problems. Developing applications that can effectively utilise the Grid, however, still remains very difficult due to the lack of high level tools to support developers.

The mostly used attempt to define Grid computing is through an analogy with the electric power evolution around 1910. The truly revolutionary development was not the discovery of electricity itself, but the electric power grid that provides standard, reliable, and low cost access to the associated transmission and distribution technologies. Similarly, the Grid research challenge is to provide standard, reliable, and low cost access to the relatively cheap computing power available nowadays.

A computational Grid was originally defined as a hardware and software infrastructure that provides dependable, consistent, pervasive, and inexpensive access to high-end computational capabilities. With the time, the Grid concept has been refined and better formulated, e.g. as a persistent infrastructure that supports computation and data intensive collaborative activities that spawn across multiple Virtual Organisations.

The natural starting point in building computational Grids is the existing world wide Internet infrastructure that aggregates a potentially unbounded number of resources. Analogous to the World Wide Web that provides ubiquitous access to the information over the Internet, the computational Grids explore new mechanisms for ubiquitous access to computational resources and quality of service beyond the best effort provided by the Internet protocol (IP).

The ***Grid Security Infrastructure (GSI)*** is the defacto standard for authentication and secure communication across applications and services. GSI has the following distinguishing characteristics that makes it suitable for being applied in Grid environments:

1. *Public key cryptography* based on private and public key pairs is the fundamental technology used for encrypting and decrypting messages;
2. *Digital signatures* are employed for insuring data integrity over the network; *X.509 certificates* are used for representing the identity of each user in the process of authentication. An X.509 certificate includes four primary pieces of information:
  - a) *Subject name* which identifies the person or the object that the certificate represents;
  - b) *Public key* that belongs to the subject;
  - c) *Certificate Authority* that signed the certificate and certifies that the public key and the subject name belong to the same trusted subject;
  - d) *Digital signature* of the trusted certificate authority;
3. *Mutual authentication* is a protocol which ensures that the two parties involved in communication identify each other and trust their certificate authorities;
4. *Secure private keys* promote the encrypted store of the user private key exclusively on the local personal computer (i.e. laptop) or on cryptographic smartcards;
5. *Single sign-on* restricts the user authentication to one single password (keyboard) specification during a working session;
6. *Proxy cryptography* creates a new private and public key-pair digitally signed by the user that temporarily represents the user's Grid identity. This allows the true private key of the user be un-encrypted for a minimum amount of time until the signed proxy is generated which minimizes the danger of loosing the identity;
7. *Proxy delegation* allows remote services behave on behalf of the client through the creation of remote proxies that impersonate the user.

Since 1995, the ***Globus Toolkit (GT)*** is the driving force in Grid computing, developing middleware technology aimed to support and ease the development of high level Grid infrastructures and applications with special focus on high performance scientific computing. The version two of GT, in short GT2, was the most successful and stable Globus release provides the following three categories of fundamental services for building higher level Grid infrastructures:

1. *Resource management services* for executing applications on remote Grid sites, which comprise:
  - a) *Grid Resource Allocation Manager (GRAM)* that provides a single GSI-enabled interface for allocating and using remote computational resources on top of existing local resource managers like Condor, Load Sharing Facility, Maui, Portable Batch System (PBS), Sun Grid Engine, or simple Unix fork system call;
  - b) *Dynamically-Updated Request Online Coallocator (DUROC)* that employs multiple GRAM services for co-allocation of several Grid sites for executing the same application instance. DUROC requires reservation functionality from the local resource manager in order to work effectively in real Grid

- environments; GRAM and DUROC use the *Resource Specification Language (RSL)* to formulate resource requirements;
2. *Information services* implemented by the *Monitoring and Discovery Service (MDS)* that comprises:
    - a) *Grid Resource Information Service (GRIS)* that provides information about a particular site using an underlying sensor like the Ganglia for machine information;
    - b) *Grid Index Information Service (GIIS)* that provides hierarchical means of aggregating multiple GRIS services for a coherent Grid system image and efficient high performance resource query support;
  3. *Data Grid services* represented by the:
    - a) *Global Access to Secondary Storage (GASS)* libraries and utilities which simplify the process of porting and running of applications in a Grid environment by installing a transparent distributed file system that eliminates the need for manual login to remote Grid sites;
    - b) *GridFTP* which is a high performance, secure, and reliable data transfer protocol optimised for high bandwidth use of wide area networks based on the highly popular FTP protocol;
    - c) *Globus Replica Catalogue* which is a mechanism for maintaining a catalogue of data set replicas;
    - d) *Globus Replica Management* which is a mechanism that ties together the Replica Catalogue and the GridFTP technologies for remote management of large data set replicas. The Globus Replica Catalogue and Replica Management services are very specific to data Grid.

GT2 on its own suffers from substantial integration and deployment problems, mostly due to its scripting or C-based interface and implementation. The Java Commodity Grid Kit (CoG) adds a layer on top of GT2 that exports a platform independent Java interface to the Globus services. GT2 and Java CoG, augmented with GSI and Web services support, represent an excellent starting point for implementing higher level Grid architectures.

A Grid site is a sequential or parallel computer accessible through one hosting environment and one single Grid Resource Allocation Manager (GRAM) service using policies established by local administration authorities usually through a local resource allocation manager.

### **2.2.2 Service Grids**

The Grid community has generally acknowledged Web services as the defacto standard technology for the realization of the service-oriented Grid architectures. However, the standard Web services technologies are designed for integration of *persistent and stateless business processes*, in contrast to the Grid services that need to model *transient and stateful Grid resources*. Examples of target stateful resources include sites with limited availability.

In this context, the *Open Grid Services Architecture (OGSA)* is the generic broad architectural model currently being defined within the Global Grid Forum that defines design mechanisms to uniformly expose Grid services semantics, to create, name, and discover transient Grid service instances, to provide location transparency and multiple protocol bindings for service instances, and to support integration with underlying native platform facilities. Extensive joint efforts in both Grid and Web communities are currently working towards defining a widely accepted standard for building OGSA compliant interoperable Grid services.

A Grid service is a Web service enhanced with standard interface support for expressing lifecycle, state, and asynchronous events required for modeling and controlling dynamic, stateful, and transient Grid resources. A Grid site can host multiple Grid services within its hosting environment that can be remotely accessed using Web services XML-based document exchange. Two persistent Grid services are required to exist in a Grid environment:

1. *Factory* for creating transient Grid service instances on arbitrary remote Grid sites;
2. *Registry* for light-weight publication, management and high throughput discovery of transient Grid services.

A service-oriented architecture is suitable for implementing Grid environments due to several significant advantages that it offers:

- it increases the *portability* and facilitates the *maintenance* of the system by isolating platform dependent services to appropriate sites accessible using a well-defined platform independent interface;
- it enables *interoperability* by providing well-defined standard network protocols for communicating with the remote services;
- it enables *light-weight clients* which are easy to be installed and managed by unexperienced users by isolating complex implementation functionality within external services;
- it *decouples* the clients from the rest of the system and allows the users to *move, share, and access* the services from different Grid locations.

## 2.3 Service-Oriented Architecture

The term SOA was first coined by Yefim Natis in one of the research papers in 1994: *SOA is a software architecture that starts with an interface definition and builds the entire application topology as a topology of interfaces, interface implementations, and interface calls.* Despite being coined much earlier, SOA started to become a buzzword only in early 2000. With the advent of Web services and WSDL compliant business process, SOA started to become popular among technology enthusiasts.

### 2.3.1 SOA – an Architectural Style

SOA is not a solution, it is a practice. A service-oriented architecture is a style of design that guides all aspects of creating and using business services throughout their lifecycle (from conception to retirement). An SOA is also a way to define and provision an IT infrastructure to allow different applications to exchange data and participate in business processes, regardless of the operating systems or programming languages underlying those applications.

An SOA can be thought of as an approach to building IT systems in which business services (i.e., the services that an organization provides to clients, customers, citizens, partners, employees, and other organizations) are the key organizing principle used to align IT systems with the needs of the business. In contrast, earlier approaches to building IT systems tended to directly use specific implementation environments such as object orientation, procedure orientation, and message orientation to solve these business problems, resulting in systems that were often tied to the features and functions of a particular execution environment technology such as CICS, IMS, CORBA, J2EE, and COM/DCOM.

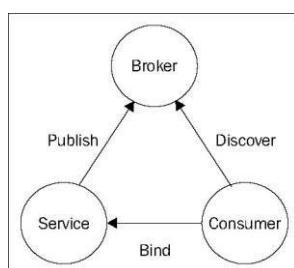
SOA as an architectural style promotes software reusability by creating reusable services. Traditional object-oriented architectures promote reusability by reusing classes or objects. However, objects are often too fine grained for effective reuse. Hence, component-oriented architectures emerged that use software components as reusable entities. These components consist of a set of related classes, their resources, and configuration information. Component-oriented architectures remain a powerful way to design software systems; however, they do not address the additional issues arising from current day enterprise environments. Today, enterprise environments are quite complex due to the use of a variety of software and hardware platforms, Internet-based distributed communication, enterprise application integration, and so on. The service-oriented architectures address these issues by using a service as a reusable entity. The services are typically coarser grained than components, and they focus on the functionality provided by their interfaces. These services communicate with each other and with end-user clients through well-defined and well-known interfaces. The communication can range from a simple passing of messages between the services to a complex scenario where a set of services together coordinate with each other to achieve a common goal. These architectures allow clients, over the network, to invoke a Web service's functionality through the service's exposed interfaces.

The fundamental of SOA is based upon: Service, Message and Dynamic discovery. In a service-oriented architecture, you have the following:

- A **service** that implements the business logic and exposes this business logic through well-defined interfaces.
- A **registry** where the service publishes its interfaces to enable clients to discover the service.
- **Clients** (including clients that may be services themselves) who discover the service using the registries and access the service directly through the exposed interfaces.

Therefore, at a high level, SOA is formed out of three **core components**:

1. Service Provider (Service)
2. Service Consumer (Consumer)
3. Directory Services (enabled by Broker)



*Fig. 2.4 SOA's core components*

From the preceding figure, we can see that:

- The service provider offers business processes in the form of services.
- The services offered by the provider are called by the consumer to achieve certain sets of business goals.
- The process of services being provided and consumed is achieved by using directory services that lie between the provider and the consumer, in the form of broker.

The service to be made available to the consumer is published to the directory services in the broker. The consumer wanting to achieve the set of business goal(s) will discover the service from the broker. If the service is found, it will bind to the service and execute the processing logic.

### 2.3.2 Service Abstraction

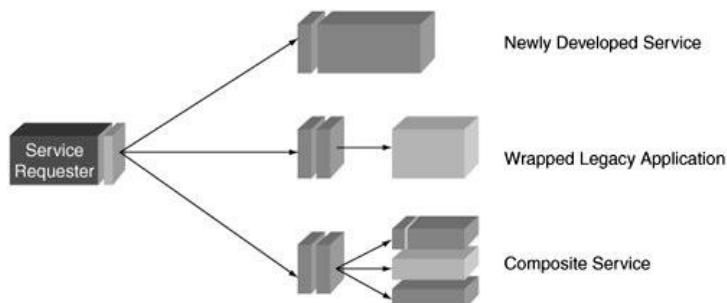
A service is a location on the network that has a machine-readable description of the messages it receives and optionally returns. A service is therefore defined in terms of the message exchange patterns it supports. A schema for the data contained in the message is used as the main part of the contract (i.e., description) established between a service requester and a service provider. Other items of metadata describe the network address for the service, the operations it supports, and its requirements for reliability, security, and transactionality.

The service implementation can be any execution environment for which Web services support is available. The service implementation is also called the executable agent. The executable agent is responsible for implementing the Web services processing model as defined in the various Web services specifications. The executable agent runs within the execution environment, which is typically a software system or programming language.

An important part of the definition of a service is that its description is separated from its executable agent. One description might have multiple different executable agents associated with it. Similarly, one agent might support multiple descriptions. The description is separated from the execution environment using a mapping layer (sometimes also called a transformation layer). The mapping layer is often implemented using proxies and stubs. The mapping layer is responsible for accepting the message, transforming the XML data to the native format, and dispatching the data to the executable agent.

Web services roles include requester and provider. The service requester initiates the execution of a service by sending a message to a service provider. The service provider executes the service upon receipt of a message and returns the results, if any are specified, to the requester. A requester can be a provider, and vice versa, meaning an execution agent can play either or both roles.

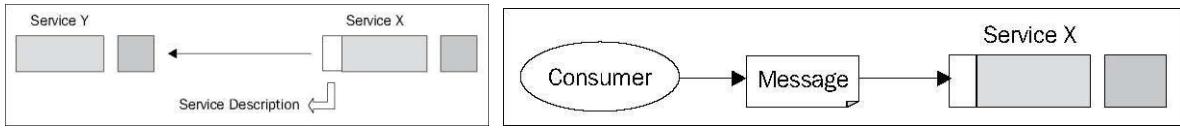
As shown in Figure 2.5 one of the greatest benefits of service abstraction is its ability to easily access a variety of service types, including newly developed services, wrapped legacy applications, and applications composed of other services (both new and legacy).



**Fig. 2.5. Requesting different types of services.**

The fundamental approach of SOA that offered the business logic to be decomposed amongst disparate services, each of which was a distinct logical unit but in entirety was part of a distributed enterprise solution. These logical units are services.

The business logic gets encapsulated in a service. A service can be an independent logical unit or it can contain in itself other set of services, as shown in Fig. 2.5. The service is used to call other sets of dependant services, to refer to those services, they must contain the *service descriptions*. The service description in its basic form contains the information of service name and location of the service being called.

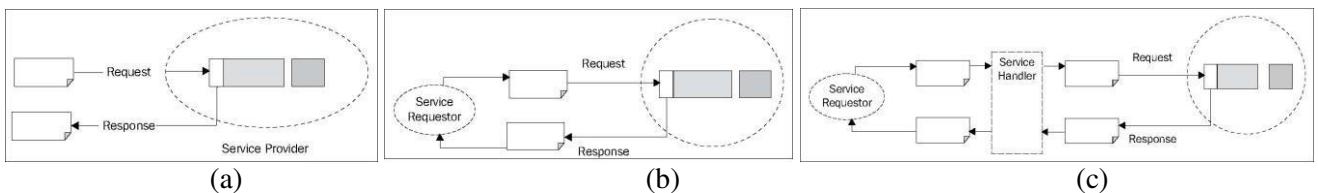


**Fig. 2.6 (a) Service to Service (b) Communication between consumer and server**

These logical units though had to adhere to certain sets of communication standards to enable information flow across the enterprise offerings in an understandable form. The information is exchanged in the form of messages from the interface designed within the system. The interface exposed by a service contains the service behavior and messaging pattern. One of the basis of SOA being platform-neutral is that messages are exchanged in XML formats so as to adhere to the concept.

'Service' as a sole unit is an independent logical unit of a business process. The business logic stands encapsulated into the service, and it interacts with the outer world through the 'interface'. The services are designed to be flexible in terms of addition of new business logic or change of logic. They should also be reusable, so that other processes can use functionality. Services are published by the 'provider' and they bind to the 'consumer' through the service 'handler'.

1. **The Service Provider:** The provider comes into action when the service is invoked. Once the service is invoked, the provider will execute the business logic. Messaging will depend upon the business logic, in case the consumer expects a message after the execution of business process, the provider will send out the reply.
2. **The Service Consumer:** The consumer would send out a message to the provider in order to access the service. This is the requester. It would either be done directly by a service-to-service call or through the directory services. Services required for processing are identified by their service descriptions. The same service can act as the provider as well as the requester of the service. But this is seldom seen in practice. Here, we have extended the above image Fig.2.7(b).
3. **The Service Handler:** The service handler acts as a collaboration agent between the provider and the consumer. The handler contains the realization logic, which will search the appropriate service provided and bind it to the consumer request. Once the service has been requested, it goes through various messaging paths and, at times, into multiple handlers to finally accomplish the logic. The handler usually routes the messages to the target system or sometimes does some processing logic before forwarding the request to target system.

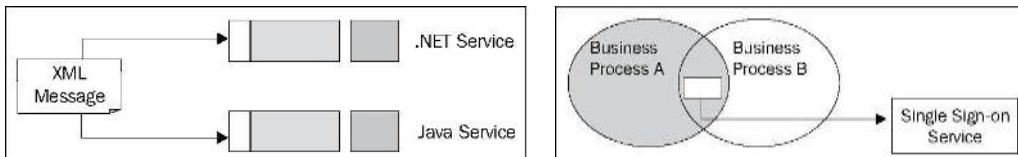


**Fig. 2.7 Requests and Responses**

### 2.3.3 SOA Objectives

The **objective** of using SOA:

- **Loose coupling:** The business process being decomposed into independent services will help in bringing down the dependencies on a single process. This in turn will help in faster processing time.
- **Platform-neutrality:** XML-based message information flow enhances the capability to achieve platform neutrality. These XML messages are based on agreed XML schema, eliminating the need to set up other messaging standards that can differ across platforms (Fig. 2.8 (a)).
- **Standards:** The message flow across the enterprise is in the form of globally accepted standards. The service only has to depend on the service descriptions without worrying about the target standards and removing the dependencies.
- **Reusability:** The business logic being divided into smaller logical units, the services can easily be re-used. These enhance the utilization of SOA-based solution, which has a cascading affect on service delivery and execution (Fig. 2.8 (b)).
- **Scalability:** Again, as the business processes are decomposed into smaller units, adding new business logic is easy to accomplish. The new logic could either be added as an extended unit of the current service, or it can also be constructed as a new service.



*Fig. 2.8 SOA Objectives: (a) Platform-neutrality (b) Reusability.*

### 2.3.4 Advantages of SOA

An important advantage of a service-oriented architecture is that it enables development of loosely-coupled applications that can be distributed and are accessible across a network. To enable this architecture, a SOA needs:

- A mechanism that enables clients to access a service and registry.
- A mechanism to enable different services to register their existence with a registry and a way for clients to look up the registry of available services. Web services are based on an architecture in which the service can be located over the network and its location is transparent, which means that clients may dynamically discover a particular service they wish to use.
- A mechanism for services to expose well-defined interfaces and a way for clients to access those interfaces.

Moreover, the followings where identifies as needs for organizations to align their business process, and design it according to service-oriented concepts, joining the SOA wagon wheel.

- **Integration:** An SOA-based solution is usually based upon the principles of inter-operability. The integration solutions thus offered are loosely coupled and less complex. At the granular level, services are being used to interact with vendors. The compounded benefit can be found in the lower cost of integration development, as we move away from proprietary integrations solutions to open standard-based solutions. The ROI can be easily measured for integration solutions as the cost per integration is drastically reduced by the use of SOA-based solution against the traditional middleware solutions. Over the period of time, organizations can move away from the current, expensive, integration solutions to SOA-based vendor-neutral integration standards. It can be achieved by standardizing the current service description and messaging solutions.
- **Business Agility:** One of the most important benefits of organizations adopting SOA is felt by the increased agility within the systems. Though agility is a non-quantifiable term, the inherent benefit is felt within the organization's hardware and software assets. The benefit in terms of software assets can be derived from SOA's ability to re-use and simplify integrations. Unlike earlier days, where development of new business process would take quite some time, the current business users will find the development period getting shortened. This makes it easy to accommodate changes, and the benefits of the same can be seen in the long term, as the enterprise solution evolves over a period of time. In terms of hardware benefits, due to the abstract use of services being loosely coupled, they can be delegated across the domain and the results can still be achieved. This helps in balancing the business processes load across the organization, and the capabilities can be utilized better. Thus, a remarkable improvement in the efficiency of business can be felt.
- **Assets Re-use:** The foremost goal of a SOA-based solution is 're-use'. Most of the earlier solutions were built-in a very tightly coupled or an isolated environment. This made it very difficult to re-use the components of the current solution. SOA-based services were built in such a manner that, though the services conformed to the current business requirement, they could still be re-used in any composite service. As a result, organizations saw the benefits of re-use in terms of a higher intial development period. But over time, the economics of re-use got better of the development span. The economics of re-use was felt in terms of faster integration and lower cost per integrations. Re-use also enabled organization to put less money into asset growth, as the current assets were being re-used effectively.
- **Increased ROI:** With proper governance and compliance in place, and a highly secured transaction environment, the adoption of SOA sees a definite increase in terms of ROI. With the integration solutions moving from expensive, tightly coupled, standard-specific, vendor dependent to being loosely coupled, vendor-neutral, open standard-based solutions, the cascading effect on ROI is seen immediately. Over time, as organizations move away from proprietary solutions to SOA-based solutions, the investment in integration assets will surely dwindle. Building solutions that are inherently re-usable helps organizations to build and market the solutions in a rapid manner. This helps organizations to improve their time-to-market, and improve efficiency with respect to customer satisfaction, service, and effective use of manpower.

### 2.3.5 Technologies used to implement SOA

As a lot of organizations move towards adopting the SOA culture, the biggest issue faced by them is the complexity of the solutions. The dismantling of the current business processes into smaller services is a huge

challenge in itself. SOA is a natural improvement over the **object-oriented (OO)** and the **component-based development (CBD)**. So, it still retains some of the flavors from each of them.

The business processes are powered by small pieces of software known as 'components'. The business logic inside the components is based on the principles of OO programming. These business processes are termed as 'services' in the analogy of SOA.

The recipe for success of any SOA solution is to ensure the classification of business processes into smaller units. You can either choose the top-down, the bottom-up, or the middle-out approach.

1. **Top-down:** In a top-down approach, the business use cases are created, which gives the specifications for the creation of services. This would ensure that the functional units are decomposed into smaller processes and then developed.
2. **Bottom-up:** Using the bottom-up approach, the current systems within the organization are studied, and suitable business processes are identified for conversion to services.
3. **Middle-out:** The middle-out approach acts as a spy, and tries to locate suitable business processes that were left out by the other two approaches.

Over the years, SOAs have been implemented using a wide variety of technologies:

- Distributed objects CORBA, J2EE, COM/DCOM.
- Message-oriented middleware (MOM) WebSphere MQ, Tibco Rendezvous.
- TP monitors CICS, IMS, Encinia, Tuxedo.
- Homegrown middleware developed in house.
- B2B platforms bXML, RosettaNet.

Of course, some of these technologies are better suited for SOAs than others. The more capabilities a technology provides that match the Web services platform, the better that technology is suited to implement an SOA.

**WebSphere MQ.** Many large organizations have created SOAs using WebSphere MQ, such as AXA Financial. AXA Financial, a \$US 7.5 billion insurance and financial services company, achieved a 300 percent return on this SOA investment using IBM's WebSphere MQ as a messaging and integration layer to connect legacy systems with front-end applications. AXA created an SOA that sits in front of all of the legacy systems and provides all of the communications between the legacy and front-end systems. AXA began developing the architecture in 1989. The SOA integration architecture currently handles more than 600,000 transactions a day. Although WebSphere MQ can be used to implement an SOA, this does not mean that all WebSphere MQ systems are service-oriented, and in fact, it is probably safe to say that only a small fraction of WebSphere MQ systems are service-oriented.

**CORBA.** Many large organizations have created SOAs using CORBA, such as Credit Suisse. Using CORBA to implement an SOA requires looking beyond CORBA's traditional role in distributed object technology and realizing that it provides many of the key ingredients of a service platform, including the following:

- CORBA is an open standard.
- CORBA supports remote method invocation (i.e., RPC calls), asynchronous messaging, and publish/subscribe communications.
- CORBA provides integrated security, naming services, transaction management, and reliable messaging.
- CORBA supports multiple programming languages.
- CORBA provides CORBA IDL, which can be used as a service definition language.
- CORBA objects can be exposed as Web services because the OMG has defined a CORBA IDL to WSDL mapping.

Of course, CORBA also has some limitations for implementing an SOA:

- CORBA is perceived as being complex.
- CORBA requires both the service requester and the service provider to be using CORBA.
- CORBA does not provide explicit support for XML and does not support loosely coupled, asynchronous exchange of business documents over the Internet.

Here is a case study of an SOA being implemented using CORBA. Credit Suisse Group is a leading global financial services company headquartered in Zurich, Switzerland. In 1997, Credit Suisse started the implementation of an SOA called the Credit Suisse Information Bus (CSIB), an integration infrastructure meant to enable service-oriented access to the back-end core application systems. The goal of the CSIB was to enable

reliable, secure, and scalable real-time request/reply interoperability between back-end systems and a variety of front-end applications based on different platforms (J2EE, C++, SmallTalk, HTML, COM, and Visual Basic). CSIB is built using CORBA, specifically, IONA's Orbix product, and it replaced an integration infrastructure based on IBM WebSphere MQ that was becoming expensive and difficult to maintain because Credit Suisse had to develop a number of layers on top of WebSphere MQ to support the request/reply paradigm and to provide error handling, data conversion, a naming service, and other services. Credit Suisse's SOA supports more than 100,000 users, including 600 business services in production. Credit Suisse's SOA has resulted in a 70 percent reuse of business services, a 73 percent cost reduction for systems development and integration, and time to market of new solutions has dramatically improved.

As with WebSphere MQ, although CORBA can be used to implement an SOA, this does not mean that all CORBA systems are service-oriented, and in fact, it is probably safe to say that only a small percentage of CORBA systems are service-oriented.

**Java and J2EE technologies** have many of the same advantages and disadvantages as CORBA when it comes to implementing an SOA. Here are some of the similarities related to SOA:

- Both are open standards.
- Both are distributed object technologies that provide excellent support for remote method invocation (i.e., RMI-IIOP and IIOP).
- Both require the service requester and the service provider to be using the same technology stack (i.e., J2EE and CORBA).
- Both provide integrated security, naming services (JNDI and CORBA Naming Service), transaction management (JTA/JTS and Object Transaction Service), and reliable messaging (JMS and CORBA Notification).
- Both J2EE EJBs and CORBA objects can be exposed as Web services. J2EE provides explicit support for Web services starting with J2EE 1.4, and the OMG has defined a CORBA IDL to WSDL mapping.

Here are some of the differences related to SOA:

- CORBA supports multiple programming languages.
- CORBA provides CORBA IDL as an explicit interface definition language.
- J2EE Web services communicate natively using XML and SOAP, whereas the CORBA WSDL mapping still communicates using CDL and IIOP.
- The Java Community Process has defined a series of APIs for manipulating XML (e.g., JAX-RPC, JAAS, JAX-B, and so on).
- J2EE has a much larger and more robust developer community.
- J2EE implementations are available from most of the major IT vendors.

As with WebSphere and CORBA, although J2EE can be used to implement an SOA, this does not mean that all J2EE systems are service-oriented, and in fact, it is probably safe to say that most J2EE applications are tightly coupled, stovepipe applications.

Although often overlooked when discussing SOA, **B2B platforms** such as ebXML and RosettaNet are ideal Web services platforms because:

- They are open standards.
- They are loosely coupled.
- They are based on XML.
- They are based on the asynchronous exchange of business documents (i.e., XML documents).
- They provide integrated mechanisms for service registration, service security, service monitoring and management, business process management, compensating transactions, and reliable messaging.

To assess the current maturity of XML Web Services for SOAs, they can be compared to earlier technologies that were successfully used to build SOAs. Table 2.1 presents a side-by-side comparison of three approaches to SOAs:

- WebSphere MQ with Custom Extensions.
- CORBA.
- XML Web Services.

From this chart, you can see that XML Web Services have many of the building blocks for creating an SOA, but the extended Web services standards are required to deliver enterprise qualities of service.

**Table 2.1. Three Approaches to SOAs**

Element of the Service Platform	WebSphere MQ with Custom Extensions	CORBA	XML Web Services
Open standards	No, proprietary	Yes OMG	Yes standards in place but fragmented across multiple organizations
Support for multiple programming languages	Yes	Yes	Yes
Loosely coupled	Yes	Mediocre	Yes
XML-based	Can be	Can be	Yes
Service Contracts			
Technology neutral service contract definition	Ad hoc text file, Word, Excel, Access	CORBA IDL	WSDL and XML Schema Proposed standards WS-Policy family
Message/parameter definition	Usually COBOL Copybook	CORBA IDL	WSDL and XML Schema
Message/parameter validation	Custom built	CORBA IDL enforces interface compiler	XML Schema and validating XML parsers
Message/parameter parsing	Custom built	Automatic and built-in via language bindings	Automatic via a wide variety of tools DOM, SAX, JAAS, JAX-RPC, etc
Service-Level Data Management			
Data typing	None	CORBA IDL	XML Schema and WSDL
Data validation	None	Methods parameter validation	XML Schema and validating XML parsers
Data query	None	None	XPath and XQuery
Data transformation	None	None	XSLT
Service Registration and Discovery			
Interface repository	None	CORBA interface repository	UDDI public or private
Naming service	None	CORBA naming service	UDDI public or private
Trading service	None	CORBA trading service	UDDI public or private
Service-Level Security			
Authentication	User id / password	IIOP/TLS	HTTP/S, WS-Security
Authorization	None	IIOP/TLS	SAML, XACML
Data privacy	None	IIOP/TLS	HTTP/S, WS-Security
Data integrity	None	IIOP/TLS	HTTP/S, WS-Security
Service-Level Interaction Patterns			
One-way, synchronous	Yes	Yes	Yes
Request/response	Yes weak support, application level management of correlated message pairs	Yes	Yes
One-way, asynchronous	Yes strong support	Yes weak support	Multiple proposed standards <ul style="list-style-type: none"> <li>• WS-Reliability</li> <li>• WS-ReliableMessaging</li> </ul>
Publish/subscribe	Yes weak support	Yes	Multiple proposed standards <ul style="list-style-type: none"> <li>• WS-Eventing</li> <li>• WS-Notification</li> </ul>
Service-Level Communication			

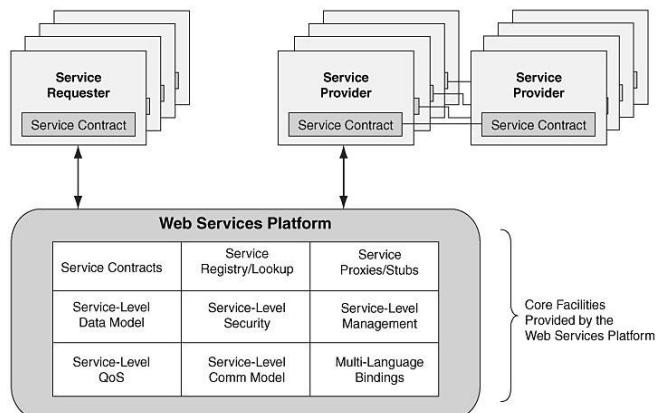
**Table 2.1. Three Approaches to SOAs**

Element of the Service Platform	WebSphere MQ with Custom Extensions	CORBA	XML Web Services
Data conversion between platforms and programming languages	Rudimentary ASCII EDCDIC	Yes	Text-based messages do not require translation
Service-Level Qualities of Service			
Session management	Yes	Yes	WS-SecureConversation, WS-Context
Load balancing	Yes	Yes	Not addressed by standards; support depends on vendor products
Guaranteed message delivery	Yes	Yes	Multiple proposed standards <ul style="list-style-type: none"> <li>• WS-Reliability</li> <li>• WS-ReliableMessaging</li> </ul>
Transaction management	WebSphere MQ's as XA-compliant transaction manager	OTS (including XA support)	Multiple proposed standards <ul style="list-style-type: none"> <li>• WS-AT, BA, C</li> <li>• WS-CAF</li> <li>• Support for both two-phase commit and compensating transactions</li> </ul>
Service-level management	None third-party products	None third-party products	Web services distributed management (WSDM) and WS-Management

### 2.3.6 SOA using Web Services

The major advantages of implementing an SOA using Web services are that Web services are pervasive, simple, and platform-neutral. Some of the important advantages of using Web services as the technology platform for an SOA are derived from the way in which the World Wide Web achieved its tremendous success; in particular, the fact that a simple document markup language approach such as HTML (or XML) can provide a powerful interoperability solution and the fact that a lightweight document transfer protocol such as HTTP can provide an effective, universal data transfer mechanism. On the Web, it doesn't matter whether the operating system is Linux, Windows, OS390, HP NonStop, or Solaris. It doesn't matter whether the Web server is Apache or IIS. It doesn't matter whether the business logic is coded in Java, C#, COBOL, Perl, or LISP. It doesn't matter whether the browser is Netscape, Internet Explorer, Mozilla, or the W3C's Amaya. All that matters is that the Web servers understand an HTTP request for an HTML file and that the browser understands how to render the HTML file into the display. Web services provide the same level of abstraction for IT systems. Similarly, all that matters for Web services is that they can understand and process an XML-formatted message received using a supported communications transport and return a reply if one is defined. Just as HTML and HTTP can be added to any computer system with a TCP connection, Web services can be added to any computer that understands XML and HTTP or XML and most other popular communications transports.

Figure 2.9 illustrates the features and capabilities of the complete Web services platform on which the broad range of SOA-based applications can be built. It includes the basic and extended Web services specifications.



**Figure 2.9. Web services platform.**

## 2.4 Ubiquitous computing

In this section we want to discuss some issues of middleware for future computing environments known under the term “ubiquitous computing” (ubicomp).

### 2.4.1 Definition

The term “ubiquitous computing” was coined by Mark Weiser in 1991. He envisioned a world of ubiquitous computers that become invisible by being embedded into the physical environment with the goal of supporting people unobtrusively in fulfilling their tasks.

One example for an application of ubiquitous computing would be a smart room allocation system, where chairs are able to sense their occupancy status and use this information to automatically derive the occupancy level of the room, which is then displayed at the electronic door plate and by a central “room finder” in the hallway.

This simple example already illustrates a number of technological features of the “ubiquitous computer.” It consists of numerous, highly specialized wireless computing devices embedded into our physical environment. These devices can perceive and control certain parameters of their physical environment and can communicate with each other. They use ergonomic, intuitive, and unobtrusive ways of interacting with people.

Recent technological advances in six important areas enable researchers already today to construct the first prototypical ubiquitous computing systems: processors, storage, wireless communication, sensors and actuators, energy supply, and the development of new materials. Moreover, researchers believe that the exponential rate of improvement of processing power, storage capacity, and communication bandwidth—which we observed over the last 30 years—will keep up for at least another 10–15 years. This observation is commonly known as “Moore’s law,” formulated by Intel founder Gordon Moore in 1965. The popular version of this rule says that the performance of computers doubles every 18 months. Traditionally, this “law” has resulted in ever faster processors, with ever increasing chip size and energy consumption. Alternatively, Moore’s law can also be “exploited” to construct processors with a more moderate performance, but with ever decreasing size and energy consumption. However, Moore’s law does not apply to the capacity of batteries and other technologies for energy storage and harvesting. Although new systems allow the extraction of energy from the environment (e.g., from mechanical vibrations or temperature differences), the amount of energy stored or harvested per device volume grows only slowly over time. Hence, the construction of energy-efficient technologies is of utmost importance, since devices for ubiquitous computing often have to be wireless. New materials (e.g., flexible displays, film batteries) will allow the construction of devices with unconventional form factors.

Overall, the general trend towards “more, smaller, cheaper, less energy” will enable the construction of future ubiquitous computing systems from a technological perspective. From a computer science perspective, new algorithms, protocols, and architectures are needed to manage and control the expected enormous amount of networked computing devices and to make sense out of the huge amount of data collected by these sensor-equipped devices.

### 2.4.2 Middleware Challenges

#### 2.4.2.1 Constrained Resources

The augmentation of artifacts with computing devices imposes constraints on the embedded devices. In order to allow an unobtrusive integration into physical objects and environments, these devices often have to be wireless and must meet certain size constraints. Limited size and energy imply that resources like computing power, memory size, communication bandwidth, and range are rather limited. Consider, for example, a matchbox-sized sensing device developed at UC Berkeley. The so-called MICA mote is equipped with an 8-bit processor with 8 MIPS, provides 8 kilobytes of RAM, 128 kilobytes of program memory, and has a communication bandwidth of 40 kilobits per second over a range of up to 30 meters. This device runs for weeks or months on a pair of AA batteries. Note that further improvements in technology will likely be used to reduce size and energy consumption, such that the performance of these devices will only increase slowly as time goes by. In contrast, current PCs are equipped with processors with hundreds of MIPS, megabytes of RAM, and a communication bandwidth of tens or hundreds of megabits per second. Although there are efforts to fit traditional middleware on resource-constrained devices, middleware such as CORBA has been designed with a PC target platform in mind.

The limited resources must be shared among various applications executing in the network and the middleware services itself. As an immediate consequence, ubicomp middleware services must be lightweight in order to fit into the constrained resources of a MICA mote and similar devices. Additionally, ubicomp middleware should provide mechanisms that help to minimize the amount of resources that are needed to accomplish a certain application task. One particularly promising approach to achieve this is to dynamically adapt the performance of hardware, algorithms, and protocols to the varying needs of the application. Interesting examples include adaptive fidelity algorithms that can be tuned to trade off output fidelity for resource usage. Another example would be to exploit application knowledge to decide when to switch off the radio for energy efficiency reasons. More concrete examples of using application knowledge appear further below.

### 2.4.2.2 Network Dynamics

A typical ubicomp application will require the collaboration of, and hence the wireless communication among, many spatially distributed devices due to the following reasons. First, ubicomp devices tend to be highly specialized: some sense environmental parameters, others extract information from the collected sensory data, and some interact with human users. A complete system often requires some or all of these functions. Second, many applications require sensory input from many spatially distributed devices (e.g., to determine the occupancy level of a room). Third, the constrained resources of individual devices often require collaboration for solving complex tasks.

Due to the limited communication range of ubicomp devices, it is unlikely that ubicomp networks would resemble mobile phone networks, where devices communicate directly with a base station, since this would require a very dense base station infrastructure. Instead, ubicomp devices will form *ad hoc networks*, where the devices act as routers, forwarding messages for their neighbors over multiple hops. More powerful devices might act as gateways that connect ad hoc network patches of ubicomp devices to an existing background infrastructure.

The topology of such ad hoc networks is subject to frequent changes due to device mobility, environmental obstructions resulting in communication failures (e.g., a truck driving by), or hardware failures (e.g., depleted batteries, stepping on a device). In sparse deployments, networks are likely to be partitioned, and devices have to operate nomadically when there are no other devices within communication range.

Ubicomp middleware has to support the robust cooperation of devices in such a highly dynamic network environment. In contrast, traditional middleware often assumes a static networking environment and considers any changes in this environment an error that is passed on to the application. CORBA, for example, throws an exception if a remote object goes temporarily offline; there are no provisions for automatically reclaiming application resources allocated for remote transactions. In the case of nomadic operation, it might be advantageous to proactively prepare for offline phases. The concept of information hoarding , for example, downloads data during online phases that might be needed later on.

In many scenarios, the application has to be able to adapt its behavior to the changing environment. Since this will be a common case, ubicomp middleware should provide adequate support mechanisms for application adaptation. In some cases it might be possible to provide automatic adaptation mechanisms that require no or little support by the application, as illustrated by the following example. Traditional communication is often address centric, where components are assigned identifiers (e.g., CORBA IORs), which are then used identify communication partners. With *data-centric communication*, distributed components are identified solely based on the function or data they provide (e.g., “some device in my vicinity that can measure temperature”). The advantage of data-centric communication is, among others, that it is able to tolerate devices going offline by transparently switching over to a device with equivalent functionality.

A further requirement on ubicomp middleware is better support for dynamic resource management. To understand this issue, note that many distributed services maintain considerable amounts of state information for each connected client. If the client disappears without notice, the allocated resources have to be reclaimed somehow. Traditional middleware such as CORBA does not adequately support such situations, leaving the task of dynamic resource management to the application. However, in ubicomp environments this is a common case that should be supported by middleware. Jini, for example, provides the *lease concept*, where resources allocated for remote peers are associated with a lease, which has to be renewed regularly. If the lease expires due to a missing renewal, the system can automatically reclaim the associated resources.

### 2.4.2.3 Scale of Deployment

Ubiquitous availability of computing resources may require a very large number of deployed computing devices. As an extreme case, consider the vision of Smart Dust, where millions of dust-grain-sized devices would be deployed in the environment in order to monitor various environmental phenomena. A single device consists of sensors, a processor, wireless communication, and energy supply. The devices are small enough to stay suspended in air, for example, to monitor weather phenomena or air quality. They could also be mixed into paint in order to coat buildings, which would allow monitoring the effects of seismic activity on the structural integrity of the buildings.

Supporting such large deployments of cooperating devices is a very challenging task. First, it is next to impossible to manually configure, maintain, fix, or upgrade individual devices due to their huge number. In the extreme case of Smart Dust, it might even be impossible to assign unique identifiers (e.g., similar to the unique MAC address of each Ethernet card) to individual nodes due to the involved production overhead. That is, starting from a totally symmetric situation (all devices are identical initially), the collection of devices must self-configure in order to achieve an operational state (e.g., set up a network topology, assign tasks to devices, collaboratively merge and evaluate collected data). Similarly, the network should be self-maintaining in order to fix node failures without manual intervention. Hence, ubicomp middleware should provide support mechanisms for self-configuration and self-maintenance.

### 2.4.2.4 Real-world Integration

By definition, ubiquitous computing devices are embedded into the physical environment, typically capturing data about their environment using attached sensors. Hence, there is a close integration of ubicomp systems with the real world. This has a number of important implications. One such implication is that physical time and location play a crucial role in ubicomp. First, it is often important to know where and when something happened. Second, time and location are crucial for correlating information from different sources. To decide whether two ubicomp devices ever met, for example, they have to share a common understanding of time and location in order to tell whether they were at the same location at the same point in time.

Establishing such a common understanding of time (i.e., time synchronization) and location (i.e., device localization) among ubicomp devices is an important middleware service. Building on that, there is also a need for services that manage spatio-temporal data. A location service, for example, maintains an up-to-date view of the current locations of devices in the network. As an extension, a history service stores location and time of past events, providing the foundation for queries like “Where did devices X and Y meet last time?”

### 2.4.3.5 Collection, Processing and Storage of Sensory Data

As a further consequence of the close integration of ubicomp systems with the real world, the collection, processing, and storage of sensory data is a core ubicomp functionality. While sensors collect rather low-level data (e.g., time series of temperature readings), applications are often interested in more high-level features (e.g., “in a conference”: used to automatically switch off mobile phones), *Context* which are also known as “context.” The derivation of context information often requires the evaluation of sensory data of various types (e.g., noise level, light intensity, air quality) originating from multiple sources. This functionality is provided by a *context service*.

In a previous section we noted the need for energy efficiency and the high energy consumption of wireless communication. As a consequence, a trivial implementation of a context service—where large amounts of raw sensory data are transmitted to a central location for processing—is often not feasible due to the resulting high energy consumption, bandwidth limitations, and scalability issues. Instead, sensory data should be preprocessed as close to its source as possible in order to reduce the amount of data that has to be transmitted. Instead of sending all the raw data to the remote application, this *in-network data aggregation* reduces communication and saves energy by transmitting compact aggregates instead of bulky raw data.

Note that these techniques to some degree blur the clear separation of communication and data processing typically found in traditional distributed systems and respective middleware. Ubicomp middleware with support for the above data reduction techniques will require means to specify application knowledge (about how to process data) and ways to inject this knowledge into the nodes of the network.

#### 2.4.2.6 Integration with Background Infrastructures

Although we noted in a previous section that ubicomp devices typically form infrastructureless ad hoc networks, it is quite likely that some of the devices will be connected to a background infrastructure such as the Internet. Some researchers believe that this will eventually lead to a global “Internet of Things” connecting smart artifacts all over the world.

There are several reasons for such an integration with background infrastructures. First, such infrastructures might be used to disseminate information (such as the room occupancy status in our introductory example) to remote destinations. Second, a background infrastructure may provide resources (e.g., computing power, storage) that are not available on typical ubicomp devices.

#### 2.4.3 Case Study: Sensor Networks

After having studied general requirements on middleware for ubiquitous computing, we will take a closer look at more concrete ubicomp middleware approaches in this section. For this, we will focus on a subarea of ubicomp known under the term “wireless sensor networks” (WSN). WSN consist of *sensor nodes*—small autonomous computing devices equipped with sensors, wireless communication capabilities, a processor, and a power supply. One prominent example of such a sensor node is the MICA sensing device that we mentioned in the previous section. Large and dense networks of these devices can be deployed unobtrusively in the physical environment in order to monitor a wide variety of real-world phenomena with unprecedented quality and scale while only marginally disturbing the observed physical processes.

In other words, wireless sensor networks provide the technological foundation for performing many “experiments” in their natural environment instead of using an artificial laboratory setting, thus eliminating many fundamental limitations of the latter. It is anticipated that a number of application domains can substantially benefit from such a technological foundation. Biologists, for example, want to monitor the behavior of animals in their natural habitats. Environmental research needs better means for monitoring environmental pollutions. Agriculture can profit from better means for observing soil quality and other parameters that influence plant growth. Geologists need better support for monitoring seismic activity and its influences on the structural integrity of buildings. And of course the military is interested in monitoring activities in inaccessible areas.

The typical usage model of a sensor network is a user specifying a high-level sensing task (e.g., “Report rooms where average noise level exceeds a certain threshold”). This task is split into many simple subtasks, which are distributed to the individual nodes of the network. These subtasks collect and preprocess low-level sensor readings. The resulting sensory data is then aggregated and processed to form a high-level sensing result that is reported back to the user.

While sensor networks can be realized by programming individual sensor nodes for a specific task, there is a strong need for abstractions that allow easy tasking of the network as a whole. Middleware for sensor networks should support such programming abstractions. Without such middleware and underlying abstractions, tasking and using a sensor network is a cumbersome and error-prone task reserved to specialists.

Now we will examine three middleware approaches with three different underlying programming abstractions. It should be emphasized that these are only first attempts whose appropriateness still has to be proven. Also, the presented systems are proofs of the concept, often only considering certain selected middleware aspects.

Likewise, operating system abstractions and concrete operating systems for sensor nodes are currently an area of active research. Hence, the functional separation and the interface between operating system and middleware is not well understood. Due to resource constraints, it is likely that operating system functionality (e.g., task and memory management) will be rather primitive compared to traditional operating systems. First operating system prototypes confirm this assumption.

##### 2.4.3.1 Databases

A number of approaches have been devised that treat the sensor network as a distributed database where users can issue SQL-like queries to have the network perform a certain sensing task. We will discuss TinyDB as a representative of this class. TinyDB supports a single “virtual” database table sensors, where each column corresponds to a specific type of sensor (e.g., temperature, light) or other source of input data (e.g., sensor node identifier, remaining battery power). Reading out the sensors at a node can be regarded as appending a new row to

sensors. The query language is a subset of SQL with some extensions.

Consider the following query example. Several rooms are equipped with multiple sensor nodes each. Each sensor node is equipped with sensors to measure the acoustic volume. The table sensors contains three columns room (i.e., the room number the sensor is in), floor (i.e., the floor on which the room is located), and volume. We can determine rooms on the 6th floor where the average volume exceeds the threshold 10 with the following query:

```
SELECT AVG(volume), room FROM sensors
WHERE floor = 6
GROUP BY room
HAVING AVG(volume) > 10
EPOCH DURATION 30s
```

The query first selects rows from sensors at the 6th floor. The selected rows are grouped by the room number. Then, the average volume of each of the resulting groups is calculated. Only groups with an average volume above 10 are kept. For each of the remaining groups, a pair of average volume and the respective room number is returned. The query is re-executed every 30 seconds, resulting in a stream of query results.

TinyDB uses a decentralized approach, where each sensor node has its own query processor that preprocesses and aggregates sensor data on its way from the sensor node to the user. Executing a query involves the following steps: First, a spanning tree of the network rooted at the user device is constructed and maintained as the network topology changes, using a controlled flooding approach. The flood messages are also used to roughly synchronize time among the nodes of the network. Second, a query is broadcast to all the nodes in the network by sending it along the tree from the root toward the leaves. During this process, a time schedule is established, such that a parent and its children agree on a time interval when the parent will listen for data from its children. At the beginning of every epoch, the leaf nodes obtain a new table row by reading out their local sensors. Then, they apply the select criteria to this row. If the criteria are fulfilled, a partial state record is created that contains all the necessary data (i.e., room number, floor number, average volume in the example). The partial state record is then sent to the parent during the scheduled time interval. The parent listens for any partial state records from its children during the scheduled interval. Then, the parent proceeds like the children by reading out its sensors, applying select criteria, and generating a partial state record if need be. Then, the parent aggregates its partial state record and the records received from its children (i.e., calculates the average volume in the example), resulting in a new partial state record. The new partial state record is then sent to the parent's parent during the scheduled interval. This process iterates up to the root of the tree. At the root, the final partial state record is evaluated to obtain the query result. The whole procedure repeats every epoch.

#### 2.4.3.2 Mobile Agents

Another class of middleware approaches is inspired by mobile code and mobile agents. There, the sensor network is tasked by injecting a program into the sensor network. This program can collect local sensor data, can statefully migrate or copy itself to other nodes, and can communicate with such remote copies. We discuss SensorWare as a representative of this class.

In SensorWare, programs are specified in Tcl, a dynamically typed, procedural programming language. The functionality specific to SensorWare is implemented as a set of additional procedures in the Tcl interpreter. The most notable extensions are the query, send, wait, and replicate commands. Query takes a sensor name (e.g., volume) and a command as parameters. One common command is value, which is used to obtain a sensor reading. Send takes a node address and a message as parameters and sends the message to the specified sensor node. Node addresses currently consist of a unique node ID, a script name, and additional identifiers to distinguish copies of the same script. The replicate command takes one or more sensor node addresses as parameters and spawns copies of the executing script on the specified remote sensor nodes. Node addresses are either unique node identifiers or “broadcast” (i.e., all nodes in transmission range). The replicate command first checks whether a remote sensor node is already executing the specified script. In this case, there are options to instruct the runtime system to do nothing, to let the existing remote script handle this additional “user,” or to create another copy of the script. In SensorWare, the occurrence of an asynchronous activity (e.g., reception of a message, expiry of a timer) is represented by a specific event each. The wait command expects a set of such event names as parameters and suspends the execution of the script until one of the specified events occurs.

### 2.4.3.3 Events

Yet another approach to sensor network middleware is based on the notion of events. Here, the application specifies interest in certain state changes of the real world (“basic events”). Upon detecting such an event, a sensor node sends an event notification toward interested applications. The application can also specify certain patterns of events (“compound events”), such that the application is only notified if occurred events match this pattern. We discuss DSWare as a representative of this class.

DSWare supports the specification and automated detection of compound events. A compound event specification contains, among others, an event identifier, a detection range specifying the geographical area of interest, a detection duration specifying the time frame of interest, a set of sensor nodes interested in this compound event, a time window  $W$ , a confidence function  $f$ , a minimum confidence  $c_{\min}$ , and a set of basic events  $E$ . The confidence function  $f$  maps  $E$  to a scalar value. The compound event is detected and delivered to the interested sensor nodes, if  $f(E) \geq c_{\min}$  and all basic events occurred within time window  $W$ .

Consider the example of detecting an explosion event, which requires the occurrence of a light event (i.e., a light flash), a temperature event (i.e., high ambient temperature), and a sound event (i.e., a bang sound) within a subsecond time window  $W$ . The confidence function is defined as  $f=0.6 \cdot B(\text{temp})+0.3 \cdot B(\text{light})+0.3 \cdot B(\text{sound})$

The function  $B$  maps an event ID to 1 if the respective event has been detected within the time window  $W$ , and to 0 otherwise. With  $c_{\min}=0.9$ , the above confidence function would trigger the explosion event if the temperature event is detected along with one or both of the light and sound events. This confidence function expresses the fact that detection of the temperature event gives us higher confidence in an actual explosion happening than the detection of the light and sound events.

Additionally, the system includes various real-time aspects, such as deadlines for reporting events, and event validity intervals.

## 2.5 Virtualization

Virtualization is one of the hottest topics in information technology today. The increasing speed and capabilities of today's x86 processors have made virtualization possible on commodity hardware, and virtualization provides an attractive way of making the most of that hardware.

Virtualization is a powerful technology that can simplify your computing infrastructure and help you get the most bang for your buck out of the latest, fastest processors, networking, and storage technologies. The bad news is that, like anything in the real world, successfully implementing, deploying, and supporting a new IT infrastructure based on virtualization requires the same level of planning and system design that any basic shift in infrastructure always has.

### 2.5.1 What Is Virtualization?

Virtualization is simply the logical separation of the request for some service from the physical resources that actually provide that service. In practical terms, virtualization provides the ability to run applications, operating systems, or system services in a logically distinct system environment that is independent of a specific physical computer system. Obviously, all of these have to be running on a certain computer system at any given time, but virtualization provides a level of logical abstraction that liberates applications, system services, and even the operating system that supports them from being tied to a specific piece of hardware. Virtualization's focus on logical operating environments rather than physical ones; it makes applications, services, and instances of an operating system portable across different physical computer systems.

The classic example of virtualization that most people are already familiar with is virtual memory, which enables a computer system to appear to have more memory than is physically installed on that system. Virtual memory is a memory-management technique that enables an operating system to see and use non contiguous segments of memory as a single, contiguous memory space. Virtual memory is traditionally implemented in an operating system by paging, which enables the operating system to use a file or dedicated portion of some storage device to save pages of memory that are not actively in use. Known as a “paging file” or “swap space”, the system can quickly transfer pages of memory to and from this area as the operating system or running applications require access to the contents of those pages. Operating systems such as Unix - like operating systems (including Linux, the \*BSD operating systems, and Mac OS X) and Microsoft Windows all use some form of virtual memory to enable the operating system and applications to access more data than would fit into physical memory.

There are many different types of virtualization, all rooted around the core idea of providing logical access to physical resources. Today, virtualization is commonly encountered in networking, storage systems, and server processes, at the operating system level and at the machine level. Xen for example supports machine - level virtualization using a variety of lever and powerful techniques.

The use of the term “virtualization” in today’s marketing literature rivals the glory days of terms such as “Internet” and “network - enabled” in the 1990s. To try to cut through the haze surrounding what is and what is not virtualization, the next few sections discuss the most common classes of virtualization and virtualization technology today.

#### 2.5.1.1 Application Virtualization

The term “application virtualization” describes the process of compiling applications into machine -independent byte code that can subsequently be executed on any system that provides the appropriate virtual machine as an execution environment. The best known example of this approach to virtualization is the byte code produced by the compilers for the Java programming language, although this concept was actually pioneered by the UCSD P - System in the late 1970s, for which the most popular compiler was the UCSD Pascal compiler. Microsoft has even adopted a similar approach in the Common Language Runtime (CLR) used by .NET applications, where code written in languages that support the CLR are transformed, at compile time, into CIL (Common Intermediate Language, formerly known as MSIL, Microsoft Intermediate Language). Like any byte code, CIL provides a platform - independent instruction set that can be executed in any environment supporting the .NET Framework.

Application virtualization is a valid use of the term “virtualization” because applications compiled into byte code become logical entities that can be executed on different physical systems with different characteristics, operating systems, and even processor architectures.

### **2.5.1.2 Desktop Virtualization**

The term “desktop virtualization” describes the ability to display a graphical desktop from one computer system on another computer system or smart display device. This term is used to describe software such as Virtual Network Computing, thin clients such as Microsoft’s Remote Desktop and associated Terminal Server products, Linux terminal servers such as the Linux Terminal Server project, NoMachine’s NX, and even the X Window System and its XDMCP display manager protocol. Many window managers, particularly those based on the X Window System, also provide internal support for multiple, virtual desktops that the user can switch between and use to display the output of specific applications. In the X Window System, virtual desktops were introduced in versions of TWM window manager, but are now available in almost every other window manager. The X Window System also supports desktop virtualization at the screen or display level, enabling window managers to use a display region that is larger than the physical size of your monitor.

Desktop virtualization is more of a bandwagon use of the term “virtualization” than an exciting example of virtualization concepts. It does indeed make the graphical console of any supported system into a logical entity that can be accessed and used on different physical computer systems, but it does so using standard client/server display software. The remote console, the operating system it is running, and the applications you execute are actually still running on a single, specific physical machine. Calling remote display software a virtualization technology are be equivalent to considering a telescope to be a set of virtual eyeballs because you can look at something far away using one.

### **2.5.1.3 Network Virtualization**

The term “network virtualization” describes the ability to refer to network resources logically rather than having to refer to specific physical network devices, configurations, or collections of related machines. There are many different levels of network virtualization, ranging from single - machine, network-device virtualization that enables multiple virtual machines to share a single physical-network resource, to enterprise - level concepts such as virtual private networks and enterprise-core and edge-routing techniques for creating subnetworks and segmenting existing networks.

Xen relies on network virtualization through the Linux bridge - utils package to enable virtual machines to appear to have unique physical addresses (Media Access Control, or MAC, addresses) and unique IP addresses. Other server-virtualization solutions, such as UML, use the Linux virtual Point - to - Point (TUN) and Ethernet (TAP) network devices to provide user - space access to the host’s network. Many advanced network switches and routers use techniques such as Virtual Routing and Forwarding (VRF), VRF - Lite, and Multi - VRF to segregate customer traffic into separately routed network segments and support multiple virtual-routing domains within a single piece of network hardware.

### **2.5.1.4 Server and Machine Virtualization**

The terms “server virtualization” and “machine virtualization” describe the ability to run an entire virtual machine, including its own operating system, on another operating system. Each virtual machine that is running on the parent operating system is logically distinct, has access to some or all of the hardware on the host system, has its own logical assignments for the storage devices on which that operating system is installed, and can run its own applications within its own operating environment.

Server virtualization is the type of virtualization technology that most people think of when they hear the term “virtualization”. Though not as common, the term “machine virtualization” uniquely identify this type of virtualization, because it more clearly differentiates the level at which virtualization is taking place — the machine itself is being virtualized — regardless of the underlying technology used. Machine virtualization is therefore the technique used by virtualization technologies such as KVM, Microsoft Virtual Server and Virtual PC, Parallels Workstation, User Mode Linux, Virtual Iron, VMware, and (of course) Xen.

In the maddening whirlwind of terms that include the word “virtual,” server virtualization is usually different from the term “virtual server” which is often used to describe both the capability of operating system servers such as e - mail and Web servers to service multiple Internet domains, and system – level virtualization techniques that are used to provide Internet service provider (ISP) users with their own virtual server machine.

The key aspect of server or machine virtualization is that different virtual machines do not share the same kernel and can therefore be running different operating systems. This differs from system – level virtualization, where

virtual servers share a single underlying kernel and provide a number of unique infrastructure, customer, and business opportunities. Some of these are:

- Running legacy software, where you depend on a software product that runs only on a specific version of a specific operating system. Being able to run legacy software and the legacy operating system that it requires is only possible on virtual systems that can run multiple operating systems.
- Software system-test and quality-assurance environments, where you need to be able to test a specific software product on many different operating systems or versions of an operating system. Server virtualization makes it easy to install and test against many different operating systems or versions of operating systems without requiring dedicated hardware for each.
- Low - level development environments, where developers may want or need to work with specific versions of tools, an operating system kernel, and a specific operating system distribution. Server virtualization makes it easy to be able to run many different operating systems and environments without requiring dedicated hardware for each.

Server and machine virtualization technologies work in several different ways. The differences between the various approaches to server or machine virtualization can be subtle, but are always significant in terms of the capabilities that they provide and the hardware and software requirements for the underlying system. The most common approaches to server and machine virtualization today are the following:

- o **Guest OS:** Each virtual server runs as a separate operating system instance within a virtualization application that itself runs on an instance of a specific operating system. Parallels Workstation, VMWare Workstation, and VMWare GSX Server are the most common examples of this approach to virtualization. The operating system on which the virtualization application is running is often referred to as the “Host OS” because it is supplying the execution environment for the virtualization application.
- o **Parallel Virtual Machine:** Some number of physical or virtual systems are organized into a single virtual machine using clustering software such as a Parallel Virtual Machine (PVM). The resulting cluster is capable of performing complex CPU and data - intensive calculations in a cooperative fashion. This is more of a clustering concept than an alternative virtualization solution.
- o **Hypervisor - based:** A small virtual machine monitor (known as a hypervisor) runs on top of your machine’s hardware and provides two basic functions. First, it identifies, traps, and responds to protected or privileged CPU operations made by each virtual machine. Second, it handles queuing, dispatching, and returning the results of hardware requests from your virtual machines. An administrative operating system then runs on top of the hypervisor, as do the virtual machines themselves. This administrative operating system can communicate with the hypervisor and is used to manage the virtual machine instances. The most common approach to hypervisor - based virtualization is known as paravirtualization, which requires changes to an operating system so that it can communicate with the hypervisor. Paravirtualization can provide performance enhancements over other approaches to server and machine virtualization, because the operating system modifications enable the operating system to communicate directly with the hypervisor, and thus does not incur some of the overhead associated with the emulation required for the other hypervisor - based machine and server technologies discussed in this section. Paravirtualization is the primary model used by Xen, which uses a customized Linux kernel to support its administrative environment, known as domain0. Xen can also take advantage of hardware virtualization to run unmodified versions of operating systems on top of its hypervisor.
- o **Full virtualization:** Very similar to paravirtualization, full virtualization also uses a hypervisor, but incorporates code into the hypervisor that emulates the underlying hardware when necessary, enabling *unmodified* operating systems to run on top of the hypervisor. Full virtualization is the model used by VMWare ESX server, which uses a customized version of Linux (known as the Service Console) as its administrative operating system.
- o **Kernel- level virtualization:** This type of virtualization does not require a hypervisor, but instead runs a separate version of the Linux kernel and an associated virtual machine as a user – space process on the physical host. This provides an easy way to run multiple virtual machines on a single host. Examples of this are User - Mode Linux (UML), which has been supported in the mainline Linux kernel for quite a while but requires a special build of the Linux kernel for guest operating systems, and Kernel Virtual Machine (KVM), which was introduced in the 2.6.20 mainline Linux kernel. UML does not require any separate administrative software in order to execute or manage its virtual machines, which can be executed from the Linux command line. KVM uses a device driver in the host ’ s kernel for communication between the main Linux kernel and the virtual machines, requires processor support for virtualization (Intel VT or AMD – v Pacifica), and uses a slightly modified QEMU process as the display and execution container for its virtual machines. In many ways, KVM ’ s kernel - level virtualization is a specialized version of full virtualization, where the Linux

kernel serves as the hypervisor, but UML and KVM are unique enough to merit their own class of server virtualization.

- **Hardware virtualization:** Very similar to both paravirtualization and full virtualization, hardware virtualization uses a hypervisor, but it is only available on systems that provide hardware support for virtualization. Hypervisor - based systems such as Xen and VMWare ESX Server, and kernel - level virtualization technologies such as KVM, can take advantage of the hardware support for virtualization that is provided on the latest generation of Intel (Intel VT, aka Vanderpool) and AMD (AMD - V, aka Pacifica) processors. Virtual machines in a hardware virtualization environment can run unmodified operating systems because the hypervisor can use the hardware 's support for virtualization to handle privileged and protected operations and hardware access requests, and to communicate with and manage the virtual machines.

Hypervisor - based virtualization is the most popular virtualization technique in use today, spanning the best - known server and machine virtualization technologies, including IBM's VM operating system, VMWare's ESX Server, Parallels Workstation, Virtual Iron products, and Xen. The use of a hypervisor was pioneered by the original commercial virtual-machine environment, IBM's CP/CMS operating system, introduced in 1966, was popularized by IBM ' s VM/370 operating system, introduced in 1970, and remains a great idea today.

In 2006, VMware proposed a generic Virtual Machine Interface (VMI) that would enable multiple hypervisor-based virtualization technologies to use a common kernel level interface. VMware and Xen agreed to work together (with others) to develop a more generic interface, known as paravirt\_ops, which is being developed by IBM, VMware, Red Hat, and XenSource. The upshot of all of this is that the eventual inclusion of the paravirt\_ops patches into the mainline kernel will enable any compliant hypervisor-based virtualization technology to work with a vanilla Linux kernel, while kernel projects such as KVM will enable users to run virtual machines, themselves running any operating system, on hardware that supports them, without requiring a hypervisor. UML will continue to enable users to run additional Linux virtual machines on a single Linux system. Though this may appear confusing, increasing richness in mainline Linux support for virtualization simply falls in the "more is better" category, and enables hypervisor based virtualization technologies to compete on their technical and administrative merits.

### 2.5.1.5 Storage Virtualization

Storage virtualization is the logical abstraction of physical storage. In conjunction with different types of filesystems, storage virtualization is the key to making flexible, expandable amounts of storage available to today's computer systems.

Storage virtualization has been around for many years, and should be familiar to anyone who has worked with RAID storage, logical volumes on systems such as Linux or AIX, or with networked filesystems such as AFS and GFS. All of these technologies combine available physical disk drives into pools of available storage that can be divided into logical sections known as volumes on which a filesystem can be created and mounted for use on a computer system. A volume is the logical equivalent of a disk partition.

The core features that make storage virtualization so attractive in today's enterprise environments is that they provide effectively infinite storage that is limited only by the number and size of drives that can be physically supported by the host system or host storage system. The reporting and discovery requirements imposed by standards such as Sarbanes - Oxley, the Department of Homeland Security, or basic corporate accountability make it important to be able to store more and more information forever. Storage virtualization enables greater amounts of physical storage to be available to individual systems, and enables existing filesystems to grow to hold that information without resorting to an administrative shotgun blast of symbolic links and inter dependent mount points for networked storage.

Technologies such as RAID (Redundant Array of Inexpensive Disks) and logical volumes as provided by the Linux LVM, LVM2, and EVMS packages are usually limited to use on the system to which the actual storage devices are physically attached. Some RAID controllers are dual - ported, allowing multiple computers access to the same volumes and associated filesystems through that RAID controller, although how well this works depends on the type of filesystem in use on the shared volume and how that filesystem is mounted on both systems.

To use a logical volume manager, you must define the disk partitions that you want to use for logical volumes, create logical volumes on that physical storage, and then create a filesystem on the logical volumes. You can then mount and use these file systems just as you would mount and use filesystems that were created on physical disk partitions.

Like standard disk controllers, RAID controllers provide block - level access to the storage devices that are attached to them, although the size of the storage available from any set of disks depends on the RAID level that is being used. The devices attached to the RAID controller are then made available to the system as though they were a single disk, which you can then partition as you wish, create filesystems on those partitions, and mount and use them just as you would use single physical partitions.

Operating systems such as Linux also support software RAID, where no physical RAID controller need be present. The software RAID system functions exactly as a hardware RAID controller would, providing block - level access to available storage, but it enforces the characteristics of different RAID levels in software rather than in hardware. Software RAID is very efficient and has only slightly lower performance than many hardware RAID controllers. Many system administrators actually prefer software RAID over hardware RAID because hardware RAID controllers are very different from manufacturer to manufacturer and even controller to controller. The failure of a RAID controller typically requires a replacement controller of the same type from the same manufacturer in order to access the data on the storage device that was attached to the failed controller. On the other hand, software RAID is completely portable across all Linux systems on which the software is installed as long as they support the same physical disk drive interfaces (IDE, EIDE, SATA, and so on).

Distributed filesystem technologies such as AFS and GFS have their own internal logical-volume creation and management mechanisms, and also make it possible to share the filesystems on these logical volumes between multiple computer systems because AFS and GFS provide locking mechanisms to synchronize simultaneous writes to shared filesystems over the network. NFS, the default Network File System for most UNIX - like operating systems, also makes it possible to share logical storage across multiple computer systems, although it does this by exporting a directory from a filesystem on the logical storage rather than by directly mounting a system - specific volume or networked filesystem. Distributed filesystems such as AFS and GFS provide filesystem - level access to logical volumes. In this, they are conceptually similar to Network Attached Storage devices, which provide filesystem - level access over a network to the filesystems that they contain.

Storage virtualization has become much more accessible across multiple computer systems with the advent of Storage Area Networks (SAN), which support block - level I/O and therefore enable multiple systems to share low - level access to various types of storage devices over the network. Most SANs use expensive, high - power network technologies such as Fibre Channel and InfiniBand to provide the high levels of throughput and general performance that are most desirable when many systems share access to block - or protocol – level networked storage.

Newer technologies such as iSCSI (Internet Small Computer Systems Interface) and AoE (ATA over Ethernet) provide less expensive mechanisms for getting block - level access to networked storage devices. As the name suggests, iSCSI supports the use of the SCSI protocol over TCP/IP networks, and requires a special type of network controller. AoE provides block - level access to suitable ATA devices using only a standard Ethernet connection. Both of these perform better on higher - bandwidth networks such as Gigabit Ethernet networks, although they are certainly usable on 100 - megabit networks. iSCSI and AoE are making networked storage a very real possibility for most of today ' s data centers and IT infrastructure of any size.

### **2.5.1.6 System - Level or Operating System Virtualization**

The system - level virtualization, often referred to as, operating system virtualization, describes various implementations of running multiple, logically distinct system environments on a single instance of an operating system kernel. System - level virtualization is based on the change root (chroot) concept that is available on all modern UNIX - like systems. During the system boot process, the kernel can use root filesystems such as those provided by initial RAM disks or initial RAM filesystems to load drivers and perform other early - stage system initialization tasks. The kernel can then switch to another root filesystem using the chroot command in order to mount an on - disk filesystem as its final root filesystem, and continue system initialization and configuration from within that filesystem. The chroot mechanism as used by system - level virtualization is an extension of this concept, enabling the system to start virtual servers with their own sets of processes that execute relative to their own filesystem root directories. Operating within the confines of their own root directories and associated filesystem prevents virtual servers from being able to access files in each others ' filesystems, and thereby provides basic protection from exploits of various server processes or the virtual server itself. Even if a chroot ' ed server is compromised, it has access only to files that are located within its own root filesystem.

The core differentiator between system - level and server virtualization is whether you can be running different operating systems on different virtual systems. If all of your virtual servers must share a single copy of an

operating system kernel, this is system - level virtualization. If different virtual servers can be running different operating systems, including different versions of a single operating system, this is server virtualization, sometimes also referred to as machine virtualization. Virtualization solutions such as FreeBSD's chroot jails, FreeVPS, Linux VServer, OpenVZ, Solaris Zones and Containers, and Virtuozzo are all examples of system - level virtualization. FreeBSD jails can run logically distinct versions of FreeBSD user - space on top of a single FreeBSD kernel, and can therefore use different instances or versions of libraries, server processes, and applications. Solaris containers and zones all share the same underlying version of Solaris, and can either use completely distinct root filesystems or share portions of a filesystem. Linux - VServer, FreeVPS, and OpenVZ can run different Linux distributions in their virtual servers, but all share the same underlying kernel.

System - level virtualization provides some significant advantages over server or machine virtualization. The key to all of these is that, because they share a single instance of an operating system kernel, system - level virtualization solutions are significantly lighter weight than the complete machines (including a kernel) required by server virtualization technologies. This enables a single physical host to support many more "virtual servers" than the number of complete virtual machines that it could support. System - level virtualization solutions such as FreeBSD's chroot jails, Linux - VServer, and FreeVPS have been used for years by businesses such as Internet Service Providers (ISPs) to provide each user with their own virtual server, in which they can have relatively complete control (and, in some cases, administrative privileges) without any chance of compromising the system's primary security configuration, system configuration files, and filesystem. System - level virtualization is therefore most commonly used for server consolidation. The primary disadvantage of system - level virtualization is that a kernel or driver problem can take down all of the system - level virtual servers supported on that system.

### 2.5.1.7 Why Virtualization Today?

Virtualization is not a new concept, and has been in use for decades in the different ways highlighted in the previous section. However, virtualization is more popular now than ever because it is now an option for a larger group of users and system administrators than ever before. There are several general reasons for the increasing popularity of virtualization:

- The power and performance of commodity x86 hardware continues to increase. Processors are faster than ever, support more memory than ever, and the latest multi - core processors literally enable single systems to perform multiple tasks simultaneously. These factors combine to increase the chance that your hardware may be under utilized. Virtualization provides an excellent way of getting the most out of existing hardware while reducing many other IT costs.
- The integration of direct support for hardware - level virtualization in the latest generations of Intel and AMD processors, motherboards, and related firmware has made virtualization on commodity hardware more powerful than ever before.
- A wide variety of virtualization products for both desktop and server systems running on commodity x86 hardware have emerged, are still emerging, and have become extremely popular. Many of these (like Xen) are open source software and are attractive from both a capability and cost perspective.

More accessible, powerful, and flexible than ever before, virtualization is continuing to prove its worth in business and academic environments all over the world. The next two sections explore some of the specific reasons why virtualization can benefit your computing infrastructure and also discuss some of the issues that you must consider before selecting virtualization as a solution to your infrastructure requirements.

### 2.5.1.8 Advantages of Virtualization

Virtualization can provide many operational and financial advantages as a key technology for both enterprise-computing and software-development environments:

1. *Better Use of Existing Hardware:* Running multiple virtual machines on your existing servers enables you to make good use of your spare processing power. Multi processor or multi - core systems can even run different virtual machines on different processors or CPU cores, taking full advantage of each portion of each processor that is available on your system. You can even get more use out of the devices, such as network interfaces, that are present on your existing servers by sharing them across your virtual machines. Running multiple virtual servers on a single physical hardware system is generally known as "server consolidation."
2. *Reduction in New Hardware Costs.* Combining server consolidation with capacity planning can reduce the number of new machines that you have to buy to support new and existing services by making better use of existing systems. In some cases, server consolidation may not eliminate the cost of new hardware, but it can simply reduce that cost. For example, buying additional memory or additional network interface cards for existing systems can enable you to expand their capabilities so that they can support additional virtual machines, without having to buy complete, new systems.

3. *Reduction in IT Infrastructure Costs.* Machine rooms have a variety of per - machine infrastructure costs that you can reduce (or at least avoid increasing) by getting more mileage out of your existing hardware rather than adding new systems. In addition to power, cooling, and space savings, reducing the number of physical machines that you manage can reduce remote-access and reliability costs by requiring fewer Keyboard - Video – Mouse (KVM) systems, fewer connections to uninterruptible power supplies, and so on. Depending on how you configure networking on the physical hardware that supports your virtual machines and the number of network interface cards installed in each system, you may even be able to simplify your network cabling and reduce the number of hubs and switches that are required in the machine room.
4. *Simplified System Administration.* Using virtualization to reduce the number of physical systems that you have to manage and maintain doesn't reduce the number of systems that you are responsible for. However, it does segment the systems that you are responsible for into two groups of systems: those that are associated with specific physical resources and those that are completely virtual. The physical systems that host your virtual machines are the primary example of the first group, but this group also includes virtual machines that make specific and unique use of physical resources such as additional network cards, specific local storage devices, and so on. Running multiple virtual machines on single physical systems makes the health of those systems more critical to your business functions and introduces some new software infrastructure for virtual machine migration or cloning in the event of emerging hardware problems. Depending on the capabilities of the monitoring package that you are running, you may be able to create separate sections or alert levels for systems with an explicit physical dependency on local hardware, systems with a dependency on centralized storage systems, and systems that are purely virtual. Finally, virtualization may enable you to streamline or simplify other time - consuming but standard system administration tasks, such as backups.
5. *Increased Uptime and Faster Failure Recovery.* Increasing the isolation of virtual machines from specific physical hardware increases system availability by increasing the portability of those virtual machines. The portability of virtual machines enables them to be migrated from one physical server to another if hardware problems arise on the first system. Adopting virtualization and a strategy for automated problem detection and virtual machine migration can lower the costs that are traditionally associated with redundancy and failover because much of the hardware that was formerly required to ensure availability by having redundant physical systems can now be provided by being able to migrate multiple virtual machines to other, suitable hardware platforms in the event of emerging problems.
6. *Simplified Capacity Expansion.* Virtual machines can be moved from one physical piece of hardware to another to enable them to benefit from hardware improvements, such as more powerful CPUs, additional CPU cores, additional memory, additional or faster network cards, and so on. Similarly, storage virtualization makes it possible to transparently increase the amount of available storage and the size of existing partitions and filesystems.
7. *Simpler Support for Legacy Systems and Applications.* By running operating systems within logical partitions (known as LPARs in mainframe - speak), customers could upgrade to a newer operating system and newer, more powerful hardware without losing the ability to run the existing software and associated operating system that their businesses depended on. Using virtualization to solve legacy software problems is a simple process. It consists of installing the appropriate legacy operating system in a virtual machine, installing the legacy software, and ensuring that the legacy software functions correctly in the new environment.
8. *Simplified System - Level Development.* Being able to restart a virtual machine to test new kernels and drivers is much faster and less of an interruption to the development process than rebooting a physical machine. This approach can also provide significant advantages for low - level debugging if you are working on a desktop system that supports virtual machines because your development environment, development system, and the virtual machine can all coexist on one desktop platform.
9. *Simplified System Installation and Deployment.* Using virtual machines can simplify deploying new systems by enabling you to use a single filesystem image as the basis for all new installations. To install a new system, you can simply create a new virtual machine by cloning that filesystem and starting a new instance of a virtual machine that uses the new filesystem. The ability to host users and customers on private virtual machines can also be used to simplify infrastructure for businesses that require large numbers of systems that are often extensively customized by their users. When using full virtual machines to deploy new systems, the ability to migrate virtual machines from one host to another can also prove an asset when you're using virtual machines as a system deployment mechanism. Finally, desktop virtualization simplifies deploying new systems by reducing the amount of software that needs to be installed locally. Increasing centralization of shared resources and the standardization of deployed systems can provide significant advantages for system administrators.

10. *Simplified System and Application Testing.* Software system test and quality assurance environments are the biggest beneficiaries of virtualization. Server virtualization is extremely time and cost - effective in such situations, reducing the amount of hardware required, and also reducing or eliminating much of the time required for system installation, re installation, and subsequent configuration. An interesting use of virtualization in system testing outside the quality assurance or system test groups is using virtualization to test new releases of an operating system and associated software.

### 2.5.1.9 Disadvantages of Virtualization

Virtualization is not a panacea for all IT woes — it is not appropriate for all scenarios, and it introduces real costs and concerns all its own:

1. *Single Point of Failure Problems*
2. *Server Sharing and Performance Issues*
3. *Per - Server Network Congestion*
4. *Increase in Networking Complexity and Debugging Time*
5. *Increased Administrative Complexity.*

### 2.5.2 Basic Approaches to Virtual Systems

The most common approaches to virtual computer systems used today are the following:

1. **Shared kernel:** A single operating system kernel supports multiple virtual systems. Each virtual system has its own root filesystem. Because all virtual machines share the same operating system kernel, the libraries and utilities executed by these virtual machines must also have been compiled for the same hardware and instruction set as the physical machine on which the virtual systems are running.
2. **Guest OS:** Virtual machines run within an application that is running as a standard application under the operating system that executes on the physical host system. This application manages the virtual machines, mediates access to the hardware resources on the physical host system, and intercepts and handles any privileged or protected instructions issued by the virtual machines. This type of virtualization typically runs virtual machines whose operating system, libraries, and utilities have been compiled for the same type of processor and instruction set as the physical machine on which the virtual systems are running. However, it can also run virtual machines, libraries, and utilities that have been compiled for other processors if the virtualization application can perform instruction-set translation or emulation, as is the case with products such as Microsoft's Virtual PC product.
3. **Hypervisor:** A hypervisor is a low - level virtual machine monitor that loads during the boot process, before the virtual machines, and runs directly on the physical hardware. The hypervisor handles requests for access to hardware resources on the physical host system, traps and handles protected or privileged instructions, and so on. Hypervisor - based virtualization runs virtual machines whose operating system, libraries, and utilities have been compiled for the same hardware and instruction set as the physical machine on which the virtual systems are running. Hypervisors are used to support virtual machines in "paravirtualization," "full virtualization," and "hardware virtualization" environments. Depending on the type of hypervisor used and the specific approach to virtualization that it takes, the source code of the operating system running in a virtual machine may need to be modified to communicate with the hypervisor.
4. **Kernel - level:** The Linux kernel runs the virtual machines, just like any other user - space process. This type of virtualization runs virtual machines whose operating system, libraries, and utilities have been compiled for the same hardware and instruction set as the Linux kernel that is running them, which was compiled for the physical machine on which the virtual systems are running.
5. **Emulation:** An emulator runs virtual machines by simulating a specific type of processor, its associated instruction set, and mandatory peripheral hardware, and can therefore run operating systems and associated software that have been compiled for processors and instruction sets other than the one used by the physical hardware on which it is running. The terms "emulation" and "server/machine virtualization" are easily confused because both of these enable multiple instances of different operating systems to run on a single host system. The key difference between the two is whether they execute virtual machines that are compiled for the native instruction set of the physical hardware on which the virtual machines are running, or those that have been compiled for some other processor and instruction set. The best - known emulation technology today is QEMU. Microsoft's Virtual PC is actually an emulation environment because it emulates the PC instruction set and hardware, enabling it to boot and run x86 operating systems such as Linux and Microsoft Windows on both x86 and PPC Macintosh platforms.

## 2.6 Cloud Computing

The general accepted definition is the following: “Cloud computing is a pay-per-use model for enabling available, convenient, on-demand network access to a shared pool of configurable computing resources (e.g., networks, servers, storage, applications, services) that can be rapidly provisioned and released with minimal management effort or service-provider interaction.”

Cloud computing gets its name as a metaphor for the Internet. Typically, the Internet is represented in network diagrams as a cloud. The cloud icon represents “all that other stuff” that makes the network work. It’s kind of like “etc.” for the rest of the solution map. It also typically means an area of the diagram or solution that is someone else’s concern.

Cloud computing model encompasses a subscription-based or pay-per-use paradigm, provides a service that can be used over the Internet, extends an IT shop’s existing capabilities and provides a return on investment.

Cloud computing is a construct that allows you to access applications that actually reside at a location other than your computer or other Internet-connected device. It is the use of computer technology that harnesses the processing power of many inter-networked computers while concealing the structure that is behind it

Cloud computing promises to cut operational and capital costs and, more importantly, let IT departments focus on strategic projects instead of keeping the datacenter running. The service is accessible via a web browser (nonproprietary) or web services API; zero capital expenditure is necessary to get started; and you pay only for what you use as you use it. The main benefit for the user consists in the fact that another company hosts your application (or suite of applications), they handle the costs of servers, they manage the software updates, and you pay for the service.

The Cloud components are:

1. Clients: mobile, terminals or regular computers (benefits: Lower hardware costs, Lower IT costs, Security, Data security, Less power consumption, Ease of repair or replacement, Less noise);
2. Data centers: collection of servers where the application to subscribe is housed. Could be a large room in the basement of your building or a room full of servers on the other side of the world. The software can be installed allowing multiple instances of virtual servers to be used. A dozen virtual servers can run on one physical server.
3. Distributed servers: servers don’t all have to be housed in the same location, they can be in geographically disparate locations. If something were to happen at one site, causing a failure, the service would still be accessed through another site. If the cloud needs more hardware, they can add them at another site.

The term services in cloud computing is the concept of being able to use reusable, fine grained components across a vendor’s network. This is widely known as “as a service.” Offerings with *as a service* as a suffix include traits like the following:

- Low barriers to entry, making them available to small businesses
- Large scalability
- Multitenancy, which allows resources to be shared by many users
- Device independence, which allows users to access the systems on different hardware

The key characteristics of Cloud computing are:

1. *On-demand self-service*: A consumer can unilaterally provision computing capabilities, such as server time and network storage, as needed without requiring human interaction with each service’s provider.
2. *Ubiquitous network access*: Capabilities are available over the network and accessed through standard mechanisms that promote use by heterogeneous thin or thick client platforms (e.g., mobile phones, laptops, and PDAs).
3. *Location-independent resource pooling*. The provider’s computing resources are pooled to serve all consumers using a multitenant model, with different physical and virtual resources dynamically assigned and reassigned according to consumer demand. The customer generally has no control over or knowledge of the exact location of the provided resources. Examples of resources include storage, processing, memory, network bandwidth, and virtual machines.

4. *Rapid elasticity.* Capabilities can be rapidly and elastically provisioned to quickly scale up, and rapidly released to quickly scale down. To the consumer, the capabilities available for rent often appear to be infinite and can be purchased in any quantity at any time.
5. *Pay per use.* Capabilities are charged using a metered, fee-for-service, or advertising-based billing model to promote optimization of resource use. Examples are measuring the storage, bandwidth, and computing resources consumed and charging for the number of active user accounts per month. Clouds within an organization accrue cost among business units and may or may not use actual currency.

There are four types of Clouds:

1. Private cloud. The cloud infrastructure is operated solely within a single organization, and managed by the organization or a third party regardless whether it is located premise or off premise. The motivation to setup a private cloud within an organization has several aspects. First, to maximize and optimize the utilization of existing in-house resources. Second, security concerns including data privacy and trust also make Private Cloud an option for many firms. Third, data transfer cost from local IT infrastructure to a Public Cloud is still rather considerable. Fourth, organizations always require full control over mission-critical activities that reside behind their firewalls. Last, academics often build private cloud for research and teaching purposes.
2. Community cloud. Several organizations jointly construct and share the same cloud infrastructure as well as policies, requirements, values, and concerns. The cloud community forms into a degree of economic scalability and democratic equilibrium. The cloud infrastructure could be hosted by a third-party vendor or within one of the organizations in the community.
3. Public cloud. This is the dominant form of current Cloud computing deployment model. The public cloud is used by the general public cloud consumers and the cloud service provider has the full ownership of the public cloud with its own policy, value, and profit, costing, and charging model. Many popular cloud services are public clouds including Amazon EC2, S3, Google AppEngine, and Force.com.
4. Hybrid cloud. The cloud infrastructure is a combination of two or more clouds (private, community, or public) that remain unique entities but are bound together by standardized or proprietary technology that enables data and application portability (e.g., cloud bursting for load-balancing between clouds). Organizations use the hybrid cloud model in order to optimize their resources to increase their core competencies by margining out peripheral business functions onto the cloud while controlling core activities on-premise through private cloud. Hybrid cloud has raised the issues of standardization and cloud interoperability that will be discussed in later sections.

There are three main delivery models of the Cloud services: Software, Platform, Infrastructure (discussed in what follows).

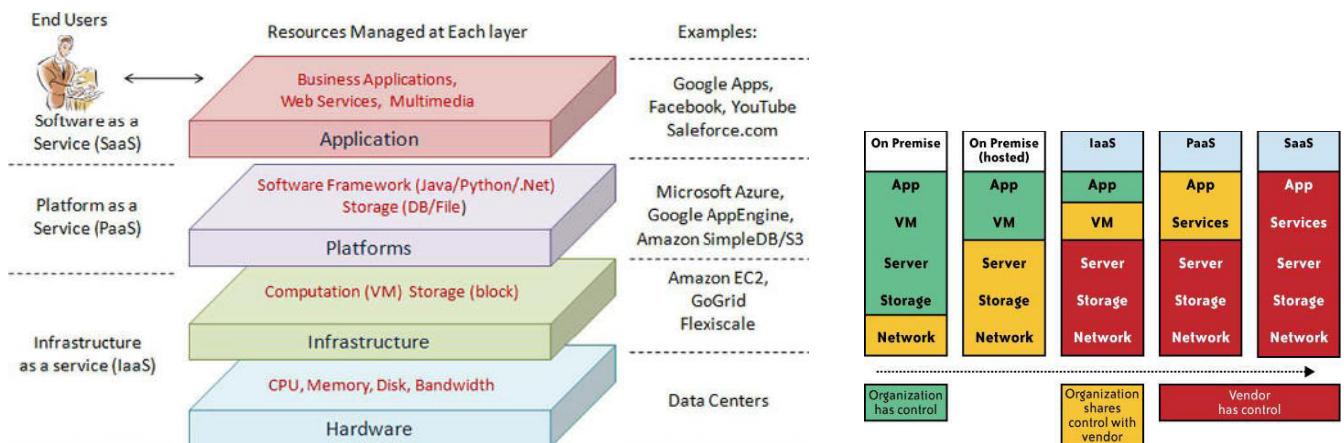


Fig. Delivery models and their effect on the actor control

## 2.6.1 Software as a Service

Software as a Service (SaaS) is the model in which an application is hosted as a service to customers who access it via the Internet. When the software is hosted off-site, the customer doesn't have to maintain it or support it. On the other hand, it is out of the customer's hands when the hosting service decides to change it. The idea is that you use the software out of the box as is and do not need to make a lot of changes or require integration to other

systems. The provider does all the patching and upgrades as well as keeping the infrastructure running. For vendors, SaaS has the appeal of providing stronger protection of their intellectual property as well as creating a continuous stream of income.

There are many types of software that lend themselves to the SaaS model. Typically, software that performs a simple task without much need to interact with other systems makes them ideal candidates for SaaS. Customers who are not inclined to perform software development but have need of high-powered applications can also benefit from SaaS. Some of these applications include:

- Customer resource management (CRM)
- Video conferencing
- IT service management
- Accounting
- Web analytics
- Web content management

Using the existing software paradigm, the user purchases a software package and license by paying a one-time fee. The software then becomes the property of the user who bought it. Support and updates are provided by the vendor under the terms of the license agreement. This can be costly if you are installing a new application on hundreds or thousands of computers. SaaS, on the other hand, has no licensing. Rather than buying the application, you pay for it through the use of a subscription, and you only pay for what you use. If you stop using the application, you stop paying.

## **2.6.2 Platform as a Service**

Platform as a Service (PaaS) is another application delivery model. PaaS supplies all the resources required to build applications and services completely from the Internet, without having to download or install software.

PaaS services include application design, development, testing, deployment, and hosting. Other services include team collaboration, web service integration, database integration, security, scalability, storage, state management, and versioning.

A downfall to PaaS is a lack of interoperability and portability among providers. That is, if you create an application with one cloud provider and decide to move to another provider, you may not be able to do so.

PaaS generally offers some support to help the creation of user interfaces, and is normally based on HTML or JavaScript. Because PaaS is expected to be used by many users simultaneously, it is designed with that sort of use in mind, and generally provides automatic facilities for concurrency management, scalability, failover, and security. PaaS also supports web development interfaces such as Simple Object Access Protocol (SOAP) and Representational State Transfer (REST), which allow the construction of multiple web services, sometimes called mashups. The interfaces are also able to access databases and reuse services that are within a private network.

PaaS is found in one of three different types of systems:

1. **Add-on development facilities** These allow existing SaaS applications to be customized. Often, PaaS developers and users are required to purchase subscriptions to the add-on SaaS application.
2. **Stand-alone environments** These environments do not include licensing, technical, or financial dependencies on specific SaaS applications and are used for general developments.
3. **Application delivery-only environments** These environments support hosting level services, like security and on-demand scalability. They do not include development, debugging, and test capabilities.

## **2.6.3 Infrastructure as a Service**

Infrastructure as a Service (IaaS) is the next form of service available in cloud computing.. Where SaaS and PaaS are providing applications to customers, IaaS doesn't. It simply offers the hardware so that your organization can put whatever they want onto it. Rather than purchase servers, software, racks, and having to pay for the datacenter space for them, the service provider rents those resources.

IaaS allows you to “rent” such resources as

- Server space
- Network equipment

- Memory
- CPU cycles
- Storage space

Additionally, the infrastructure can be dynamically scaled up or down, based on the application resource needs. Further, multiple tenants can be on the equipment at the same time. Resources are typically billed based on a utility computing basis, so providers charge by how many resources are consumed.

IaaS involves several pieces:

- **Service level agreements** This is an agreement between the provider and client, guaranteeing a certain level of performance from the system.
- **Computer hardware** These are the components whose resources will be rented out. Service providers often have this set up as a grid for easier scalability.
- **Network** This includes hardware for firewalls, routers, load balancing, and so on.
- **Internet connectivity** This allows clients to access the hardware from their own organizations.
- **Platform virtualization environment** This allows the clients to run the virtual machines they want.
- **Utility computing billing** Typically set up to bill customers based on how many system resources they use.

#### **2.6.4 Anything as a Service**

Particular cases of the above three delivery models can be considered, like:

- Storage as a Service: known as disk space on demand, is the ability to leverage storage that physically exists at a remote site but is logically a local storage resource to any application that requires storage.
- Database as a Service: provides the ability to leverage the services of a remotely hosted database, sharing it with other users and having it logically function as if the database were local
- Communication as a Service: an outsourced enterprise communications solution. Providers of this type of cloud-based solution are responsible for the management of hardware and software. Delivers: Voice over IP (VoIP) services, Instant Messaging (IM), and video conferencing capabilities. Advanced features: chat, multimedia conferencing, Microsoft Outlook integration, real-time presence, “soft” phones (software-based telephones), video calling, unified messaging and mobility etc
- Monitoring as a Service: outsourced provisioning of security, primarily on business platforms that leverage the Internet to conduct business;
- Testing as a Service: ability to test local or cloud-delivered systems using testing software and services that are remotely hosted
- Process as a Service: remote resource that can bind many resources together, e.g. services & data hosted within the same Cloud resource or remotely to create business processes;
- Information as a Service: refers to the ability to consume any type of remotely hosted information (e.g. stock price information, address validation, credit reporting)
- Identity as a Service: Offers a digital identity—a set of bytes—to describe the user. Based on this information, the application can determine who the user is and what he or she is allowed to do.
- Application as a Service
- Integration as a Service: the ability to deliver a complete integration stack from the cloud including interfacing with applications, semantic mediation, flow control, integration design, and so on. Includes most of the features and functions found within traditional enterprise application integration technology, but delivered as a service;
- Governance as a Service: any on-demand service that provides the ability to manage one or more cloud services;
- Security as a Service: the ability to deliver core security services remotely over the Internet;
- Backup as a Service: online backup SaaS.

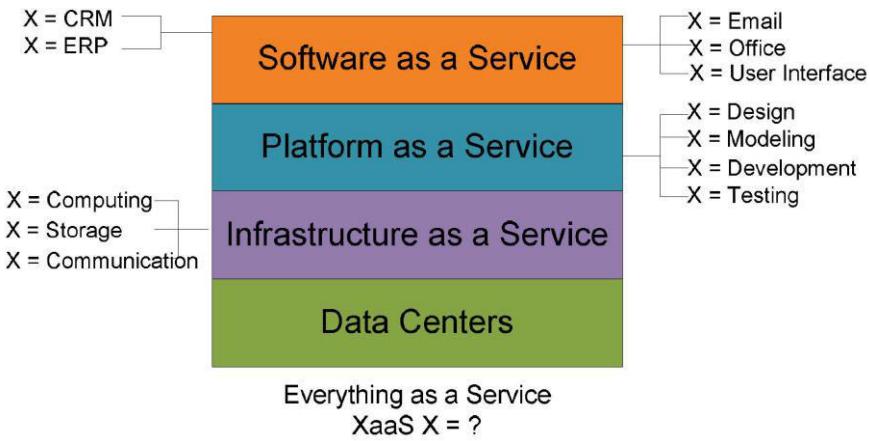


Figure: Examples of correlation between XaaS and the three basic delivery model

## 2.6.4 Pioneers

Amazon was one of the first companies to offer cloud services to the public, and they are very sophisticated. Amazon offers a number of cloud services, including:

- **Elastic Compute Cloud (EC2)** Offers virtual machines and extra CPU cycles for your organization.
- **Simple Storage Service (S3)** Allows you to store items up to 5GB in size in Amazon's virtual storage service.
- **Simple Queue Service (SQS)** Allows your machines to talk to each other using this message-passing API.
- **SimpleDB** A web service for running queries on structured data in real time. This service works in close conjunction with Amazon Simple Storage Service (Amazon S3) and Amazon Elastic Compute Cloud (Amazon EC2), collectively providing the ability to store, process, and query data sets in the cloud.

These services can be difficult to use, because they have to be done through the command line. That said, if you are used to working in a command-line environment, you shouldn't have much trouble using the services.

Amazon's virtual machines are versions of Linux distributions, so those who are experienced with Linux will be right at home. In fact, applications can be written on your own machine and then uploaded to the cloud.

Amazon is the most extensive cloud service to date. You can see more about Amazon's cloud services at <http://aws.amazon.com>.

In stark contrast to Amazon's offerings is Google's App Engine. On Amazon you get root privileges, but on App Engine, you can't write a file in your own directory. Google removed the file write feature out of Python as a security measure, and to store data you must use Google's database. Google offers online documents and spreadsheets, and encourages developers to build features for those and other online software, using its Google App Engine. Google reduced the web applications to a core set of features, and built a good framework for delivering them. Google also offers handy debugging features. Groups and individuals will likely get the most out of App Engine by writing a layer of Python that sits between the user and the database. Look for Google to add more features to add background processing services. It can be found online at [code.google.com/appengine/](http://code.google.com/appengine/).

Microsoft's cloud computing solution is called Windows Azure, an operating system that allows organizations to run Windows applications and store files and data using Microsoft's datacenters. It's also offering its Azure Services Platform, which are services that allow developers to establish user identities, manage workflows, synchronize data, and perform other functions as they build software programs on Microsoft's online computing platform.

Key components of Azure Services Platform include

- **Windows Azure** Provides service hosting and management and low-level scalable storage, computation, and networking.

- **Microsoft SQL Services** Provides database services and reporting.
- **Microsoft .NET Services** Provides service-based implementations of .NET Framework concepts such as workflow.
- **Live Services** Used to share, store, and synchronize documents, photos, and files across PCs, phones, PC applications, and web sites.
- **Microsoft SharePoint Services and Microsoft Dynamics CRM Services** Used for business content, collaboration, and solution development in the cloud.

# PART III.

## DISTRIBUTED PROGRAMMING IN JAVA

## 3.1 Modern Technologies for Distributed Applications

The early years of distributed application architecture were dominated by two-tier business applications (client-server architecture). In a two-tier architecture model, the first (upper) tier handles the presentation and business logic of the user application (client), and the second/lower tier handles the application organization and its data storage (server). This approach is commonly called **client-server** applications architecture. Generally, the server in a client/server application model is a database server that is mainly responsible for the organization and retrieval of data. The application client in this model handles most of the business processing and provides the graphical user interface of the application. It is a very popular design in business applications where the user interface and business logic are tightly coupled with a database server for handling data retrieval and processing. For example, the client-server model has been widely used in enterprise resource planning (ERP), billing, and Inventory application systems where a number of client business applications residing in multiple desktop systems interact with a central database server.

Some of the common limitations of the client-server application model are as follows:

- Complex business processing at the client side demands robust client systems.
- Security is more difficult to implement because the algorithms and logic reside on the client side making it more vulnerable to hacking.
- Increased network bandwidth is needed to accommodate many calls to the server, which can impose scalability restrictions.
- Maintenance and upgrades of client applications are extremely difficult because each client has to be maintained separately.
- Client-server architecture suits mostly database-oriented standalone applications and does not target robust reusable component-oriented applications.

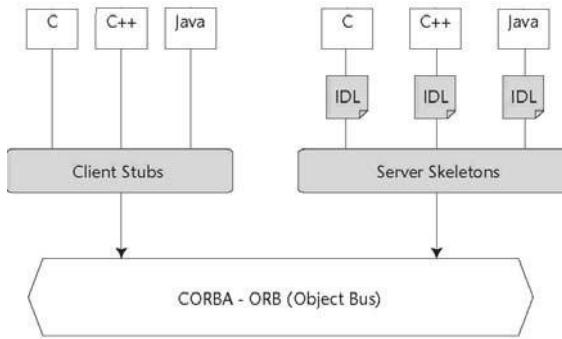
### 3.1.1 CORBA

The Common Object Request Broker Architecture (CORBA) is an industry wide, open standard initiative, developed by the Object Management Group (OMG) for enabling distributed computing that supports a wide range of application environments. OMG is a nonprofit consortium responsible for the production and maintenance of framework specifications for distributed and interoperable object-oriented systems.

CORBA differs from the traditional client/server model because it provides an object-oriented solution that does not enforce any proprietary protocols or any particular programming language, operating system, or hardware platform. By adopting CORBA, the applications can reside and run on any hardware platform located anywhere on the network, and can be written in any language that has mappings to a neutral interface definition called the Interface Definition Language (IDL). An IDL is a specific interface language designed to expose the services (methods/functions) of a CORBA remote object. CORBA also defines a collection of system-level services for handling low-level application services like life-cycle, persistence, transaction, naming, security, and so forth. Initially, CORBA 1.1 was focused on creating component level, portable object applications without interoperability. The introduction of CORBA 2.0 added interoperability between different ORB vendors by implementing an Internet Inter-ORB Protocol (IIOP). The IIOP defines the ORB backbone, through which other ORBs can bridge and provide interoperation with its associated services.

In a CORBA-based solution, the Object Request Broker (ORB) is an object bus that provides a transparent mechanism for sending requests and receiving responses to and from objects, regardless of the environment and its location. The ORB intercepts the client's call and is responsible for finding its server object that implements the request, passes its parameters, invokes its method, and returns its results to the client. The ORB, as part of its implementation, provides interfaces to the CORBA services, which allows it to build custom-distributed application environments.

Figure 3.1 illustrates the architectural model of CORBA with an example representation of applications written in C, C++, and Java providing IDL bindings.



**Fig. 3.1. An example of the CORBA architectural model**

The CORBA architecture is composed of the following components:

**IDL.** CORBA uses IDL contracts to specify the application boundaries and to establish interfaces with its clients.

The IDL provides a mechanism by which the distributed application component's interfaces, inherited classes, events, attributes, and exceptions can be specified in a standard definition language supported by the CORBA ORB.

**ORB.** It acts as the object bus or the bridge, providing the communication infrastructure to send and receive request/responses from the client and server. It establishes the foundation for the distributed application objects, achieving interoperability in a heterogeneous environment.

Some of the distinct advantages of CORBA over a traditional client/server application model are as follows:

**OS and programming-language independence.** Interfaces between clients and servers are defined in OMG IDL, thus providing the following advantages to Internet programming: Multi-language and multi-platform application environments, which provide a logical separation between interfaces and implementation.

**Legacy and custom application integration.** Using CORBA IDL, developers can encapsulate existing and custom applications as callable client applications and use them as objects on the ORB.

**Rich distributed object infrastructure.** CORBA offers developers a rich set of distributed object services, such as the Lifecycle, Events, Naming, Transactions, and Security services.

**Location transparency.** CORBA provides location transparency: An object reference is independent of the physical location and application level location. This allows developers to create CORBA-based systems where objects can be moved without modifying the underlying applications.

**Network transparency.** By using the IIOP protocol, an ORB can inter-connect with any ORB located elsewhere on the network. **Remote callback support.** CORBA allows objects to receive asynchronous event notification from other objects.

**Dynamic invocation interface.** CORBA clients can both use static and dynamic methods invocations. They either statically define their method invocations through stubs at compile time, or have the opportunity to discover objects' methods at runtime. With those advantages, some key factors, which affected the success of CORBA evident while implementing CORBA-based distributed applications, are as follows:

**High initial investment.** CORBA-based applications require huge investments in regard to new training and the deployment of architecture, even for small-scale applications.

**Availability of CORBA services.** The Object services specified by the OMG are still lacking as implementation products.

**Scalability.** Due to the tightly coupled nature of the connection-oriented CORBA architecture, very high scalability expected in enterprise applications may not be achieved.

However, most of those disadvantages may be out of date today. The Internet community for the development of Intranet and Extranet applications has acknowledged using CORBA with IIOP and Java as their tools of choice. JDK includes a full-featured CORBA implementation and also a limited set of services.

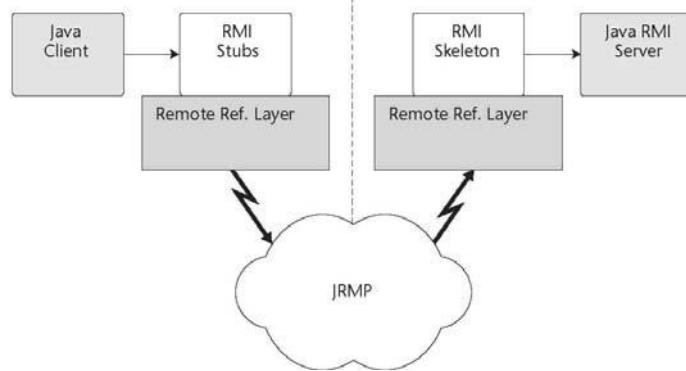
### 3.1.2 Java RMI

Java RMI was developed by Sun Microsystems as the standard mechanism to enable distributed Java objects-based application development using the Java environment. RMI provides a distributed Java application environment by calling remote Java objects and passing them as arguments or return values. It uses Java object serialization—a lightweight object persistence technique that allows the conversion of objects into streams.

Before RMI, the only way to do inter-process communications in the Java platform was to use the standard Java network libraries. Though the java.net APIs provided sophisticated support for network functionalities, they were not intended to support or solve the distributed computing challenges. Java RMI uses Java Remote Method Protocol (JRMP) as the inter-process communication protocol, enabling Java objects living in different Java Virtual Machines (VMs) to transparently invoke one another's methods. Because these VMs can be running on different computers anywhere on the network, RMI enables object-oriented distributed computing. RMI also uses a reference-counting garbage collection mechanism that keeps track of external live object references to remote objects (live connections) using the virtual machine. When an object is found unreferenced, it is considered to be a weak reference and it will be garbage collected.

In RMI-based application architectures, a registry (rmiregistry)-oriented mechanism provides a simple non-persistent naming lookup service that is used to store the remote object references and to enable lookups from client applications. The RMI infrastructure based on the JRMP acts as the medium between the RMI clients and remote objects. It intercepts client requests, passes invocation arguments, delegates invocation requests to the RMI skeleton, and finally passes the return values of the method execution to the client stub. It also enables callbacks from server objects to client applications so that the asynchronous notifications can be achieved.

Figure 3.2 depicts the architectural model of a Java RMI-base application solution.



**Figure 3.2 A Java RMI architectural model.**

The Java RMI architecture is composed of the following components:

**RMI client.** The RMI client, which can be a Java applet or a stand-alone application, performs the remote method invocations on a server object. It can pass arguments that are primitive data types or serializable objects.

**RMI stub.** The RMI stub is the client proxy generated by the rmi compiler (*rmic* provided along with Java developer kit—JDK) that encapsulates the network information of the server and performs the delegation of the method invocation to the server. The stub also marshals the method arguments and unmarshals the return values from the method execution.

**RMI infrastructure.** The RMI infrastructure consists of two layers: the remote reference layer and the transport layer. The remote reference layer separates out the specific remote reference behavior from the client stub. It handles certain reference semantics like connection retries, which are unicast/multicast of the invocation requests. The transport layer actually provides the networking infrastructure, which facilitates the actual data transfer during method invocations, the passing of formal arguments, and the return of back execution results.

**RMI skeleton.** The RMI skeleton, which also is generated using the RMI compiler (*rmic*) receives the invocation requests from the stub and processes the arguments (unmarshalling) and delegates them to the RMI server. Upon successful method execution, it marshals the return values and then passes them back to the RMI stub via the RMI infrastructure.

**RMI server.** The server is the Java remote object that implements the exposed interfaces and executes the client requests. It receives incoming remote method invocations from the respective skeleton, which passes the parameters after unmarshalling. Upon successful method execution, return values are sent back to the skeleton, which passes them back to the client via the RMI infrastructure.

Developing distributed applications in RMI is simpler than developing with Java sockets because there is no need to design a protocol, which is a very complex task by itself. RMI is built over TCP/IP sockets, but the added advantage is that it provides an object-oriented approach for inter-process communications. Java RMI provides the

Java programmers with an efficient, transparent communication mechanism that frees them of all the application-level protocols necessary to encode and decode messages for data exchange. RMI enables distributed resource management, best processing power usage, and load balancing in a Java application model. RMI-IIOP (RMI over IIOP) is a protocol that has been developed for enabling RMI applications to interoperate with CORBA components. Although RMI had inherent advantages provided by the distributed object model of the Java platform, it also had some limitations:

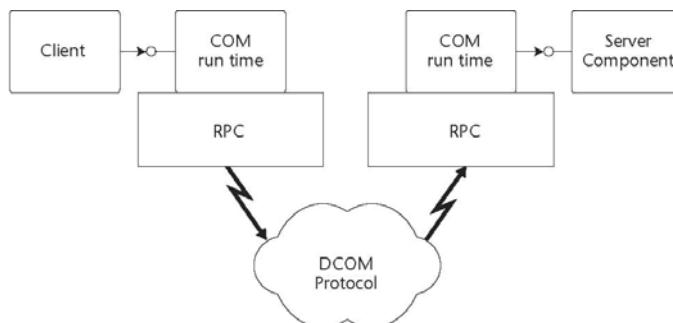
- RMI is limited only to the Java platform. It does not provide language independence in its distributed model as targeted by CORBA.
- RMI-based application architectures are tightly coupled because of the connection-oriented nature. Hence, achieving high scalability in such an application model becomes a challenge.
- RMI does not provide any specific session management support. In a typical client/server implementation, the server has to maintain the session and state information of the multiple clients who access it. Maintaining such information within the server application with-out a standard support is a complex task.

In spite of some of its limitations, RMI and RMI-IIOP has become the core of the J2EE architectural model due to its widespread acceptance in the Java distributed computing paradigm and rich features.

### **3.1.3 Microsoft DCOM**

The Microsoft Component Object Model (COM) provides a way for Windows-based software components to communicate with each other by defining a binary and network standard in a Windows operating environment. COM evolved from OLE (Object Linking and Embedding), which employed a Windows registry-based object organization mechanism. COM provides a distributed application model for ActiveX components.

As a next step, Microsoft developed the Distributed Common Object Model (DCOM) as its answer to the distributed computing problem in the Microsoft Windows platform. DCOM enables COM applications to communicate with each other using an RPC mechanism, which employs a DCOM protocol on the wire.



**Fig. 3.3. Basic architectural model of Microsoft DCOM**

Figure 3.3 shows an architectural model of DCOM. DCOM applies a skeleton and stub approach whereby a defined interface that exposes the methods of a COM object can be invoked remotely over a network. The client application will invoke methods on such a remote COM object in the same fashion that it would with a local COM object. The stub encapsulates the network location information of the COM server object and acts as a proxy on the client side. The servers can potentially host multiple COM objects, and when they register themselves against a registry, they become available for all the clients, who then discover them using a lookup mechanism.

DCOM is quite successful in providing distributed computing support on the Windows platform. But, it is limited to Microsoft application environments.

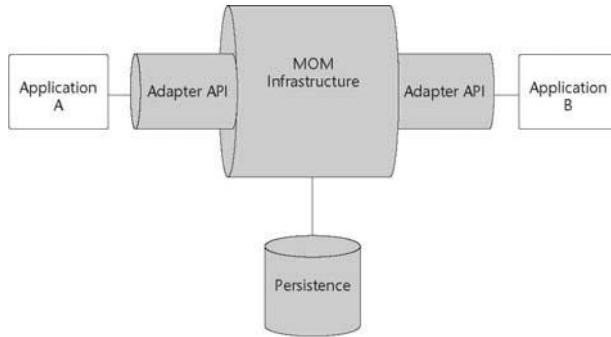
The following are some of the common limitations of DCOM:

- Platform lock-in
- State management
- Scalability
- Complex session management issues

### 3.1.4 Message-Oriented Middleware (MOM)

Although CORBA, RMI, and DCOM differ in their basic architecture and approach, they adopted a tightly coupled mechanism of a synchronous communication model (request/response). All these technologies are based upon binary communication protocols and adopt tight integration across their logical tiers, which is susceptible to scalability issues.

Message-Oriented Middleware (MOM) is based upon a loosely coupled asynchronous communication model where the application client does not need to know its application recipients or its method arguments. MOM enables applications to communicate indirectly using a messaging provider queue. The application client sends messages to the message queue (a message holding area), and the receiving application picks up the message from the queue. In this operation model, the application sending messages to another application continues to operate without waiting for the response from that application.



*Fig. 3.4. A typical MOM-based architectural model*

Figure 3.4 illustrates a high-level MOM architecture showing application-to-application connectivity. In MOM-based architecture, applications interacting with its messaging infrastructure use custom adapters. Client applications communicate with the underlying messaging infrastructure using these adapters for sending and receiving messages. For reliable message delivery, messages can be persisted in a database/file system as well.

Some of the widely known (in 2003) MOM-based technologies are SunONE Message Queue, IBM MQSeries, TIBCO, SonicMQ, and Microsoft Messaging Queue (MSMQ). The Java Platform provides a Java-based messaging API (JMS-Java Message Service), which is developed as part of the Sun Java Community Process (JCP) and also is currently part of the J2EE 1.3 specifications. All the leading MOM vendors like SunONE, TIBCO, IBM, BEA, Talarian, Sonic, Fiorano, and Spiritwave support the JMS specifications. JMS provides Point-to-Point and Publish/Subscribe messaging models with the following features: complete transactional capabilities, reliable message delivery, security.

Some of the common challenges while implementing a MOM-based application environment are the followings:

- Most of the standard MOM implementations have provided native APIs for communication with their core infrastructure. This has affected the portability of applications across such implementations and has led to a specific vendor lock-in.
- The MOM messages used for integrating applications are usually based upon a proprietary message format without any standard compliance.

Adopting a JMS-based communication model enables a standardized way to communicate with a MOM provider without having to lock in to any specific vendor API. It leverages the use of open standards 1, thus enhancing the flexibility in connecting together diverse applications.

### 3.1.5 Common Challenges in Distributed Computing

Distributed computing technologies like CORBA, RMI, and DCOM have been quite successful in integrating applications within a homogenous environment inside a local area network. As the Internet becomes a logical solution that spans and connects the boundaries of businesses, it also demands the interoperability of applications across networks. This section discusses some of the common challenges noticed in the CORBA-, RMI-, and

DCOM-based distributed computing solutions:

- Maintenance of various versions of stubs/skeletons in the client and server environments is extremely complex in a heterogeneous network environment.
- Quality of Service (QoS) goals like Scalability, Performance, and Availability in a distributed environment consume a major portion of the application's development time.
- Interoperability of applications implementing different protocols on heterogeneous platforms almost becomes impossible. For example, a DCOM client communicating to an RMI server or an RMI client communicating to a DCOM server.
- Most of these protocols are designed to work well within local networks. They are not very firewall friendly or able to be accessed over the Internet.

The biggest problem with application integration with this tightly coupled approach spearheaded by CORBA, RMI, and DCOM was that it influenced separate sections of the developer community who were already tied to specific platforms. Microsoft Windows platform developers used DCOM, while UNIX developers used CORBA or RMI. There was no big effort in the community to come up with common standards that focused on the interoperability between these diverse protocols, thus ignoring the importance, and hence, the real power of distributed computing.

### ***3.1.6 The Role of J2EE and XML in Distributed Computing***

The emergence of the Internet has helped enterprise applications to be easily accessible over the Web without having specific client-side software installations. In the Internet-based enterprise application model, the focus was to move the complex business processing toward centralized servers in the back end.

The first generation of Internet servers was based upon Web servers that hosted static Web pages and provided content to the clients via HTTP (HyperText Transfer Protocol). HTTP is a stateless protocol that connects Web browsers to Web servers, enabling the transportation of HTML content to the user. With the high popularity and potential of this infrastructure, the push for a more dynamic technology was inevitable. This was the beginning of server-side scripting using technologies like CGI, NSAPI, and ISAPI. With many organizations moving their businesses to the Internet, a whole new category of business models like business-to-business (B2B) and business-to-consumer (B2C) came into existence.

This evolution lead to the specification of J2EE architecture, which promoted a much more efficient platform for hosting Web-based applications. J2EE provides a programming model based upon Web and business components that are managed by the J2EE application server. The application server consists of many APIs and low-level services available to the components. These low-level services provide security, transactions, connections and instance pooling, and concurrency services, which enable a J2EE developer to focus primarily on business logic rather than plumbing.

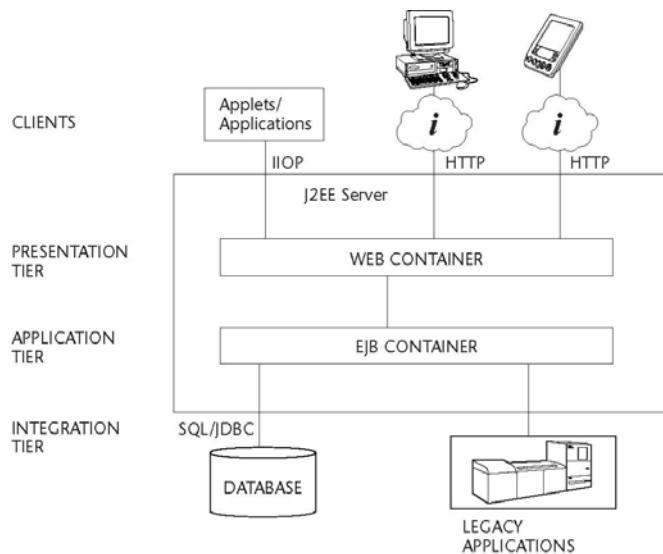
The power of Java and its rich collection of APIs provided the perfect solution for developing highly transactional, highly available and scalable enterprise applications. Based on many standardized industry specifications, it provides the interfaces to connect with various back-end legacy and information systems. J2EE also provides excellent client connectivity capabilities, ranging from PDA to Web browsers to Rich Clients (Applets, CORBA applications, and Standard Java Applications).

A typical J2EE architecture is physically divided in to three logical tiers, which enables clear separation of the various application components with defined roles and responsibilities. Figure 3.5 shows the various components of the J2EE architecture. The following is a breakdown of functionalities of those logical tiers:

Presentation tier. The Presentation tier is composed of Web components, which handle HTTP requests/responses, Session management, Device independent content delivery, and the invocation of business tier components.

Application tier. The Application tier (also known as the Business tier) deals with the core business logic processing, which may typically deal with workflow and automation. The business components retrieve data from the information systems with well-defined APIs provided by the application server.

Integration tier. The Integration tier deals with connecting and communicating to back-end Enterprise Information Systems (EIS), database applications and legacy applications, or mainframe applications.



**Fig. 3.5. J2EE application architecture**

With its key functionalities and provisions for partitioning applications into logical tiers, J2EE has been highly adopted as the standard solution for developing and deploying mission critical Web-based applications. The power of J2EE-based applications would be tremendous, if it is enabled to interoperate with other potential J2EE-deployed applications. This enables business components across networks to negotiate among them and interact without human interaction. It also enables the realization of syndication and collaboration of business processes across the Internet by enabling them to share data and component-level processes in real time. This phenomenon is commonly referred to as business-to-business (B2B) communication.

The emergence of the Extensible Markup Language (XML) for defining portable data in a structured and self-describing format is embraced by the industry as a communication medium for electronic data exchange. Using XML as a data exchange mechanism between applications promotes inter-operability between applications and also enhances the scalability of the underlying applications. Combining the potential of a J2EE platform and XML offers a standard framework for B2B and inter-application communication across networks.

With J2EE enabling enterprise applications to the Internet and XML acting as a “glue” bridges these discrete J2EE-based applications by facilitating them to interoperate with each other. XML, with its incredible flexibility, also has been widely adopted and accepted as a standard by major vendors in the IT industry, including Sun, IBM, Microsoft, Oracle, and HP. The combination of these technologies offers more promising possibilities in the technology sector for providing a new way of application-to-application communication on the Internet. It also promotes a new form of the distributed computing technology solution referred to as Web services.

### 3.1.7 Web Services

Today, the adoption of the Internet and enabling Internet-based applications has created a world of discrete business applications, which co-exist in the same technology space but without interacting with each other. The increasing demands of the industry for enabling B2B, application-to-application (A2A), and inter-process application communication has led to a growing requirement for service-oriented architectures. Enabling service-oriented applications facilitates the exposure of business applications as service components enable business applications from other organizations to link with these services for application interaction and data sharing without human intervention. By leveraging this architecture, it also enables interoperability between business applications and processes. By adopting Web technologies, the service-oriented architecture model facilitates the delivery of services over the Internet by leveraging standard technologies such as XML. It uses platform-neutral standards by exposing the underlying application components and making them available to any application, any platform, or any device, and at any location. Today, this phenomenon is well adopted for implementation and is commonly referred to as Web services. Although this technique enables communication between applications with the addition of service activation technologies and open technology standards, it can be leveraged to publish the services in a register of yellow pages available on the Internet. This will further redefine and transform the way businesses communicate over the Internet. This promising new technology sets the strategic vision of the next generation of virtual business models and the unlimited potential for organizations doing business collaboration and business process management over the Internet.

## 3.2 Sockets in Java

### 3.2.1 Networks, Packets, and Protocols

Before we delve into the details of sockets, however, it is worth taking a brief look at the big picture of networks and protocols to see where our code will fit in. Our goal here is *not* to teach you how networks and TCP/IP work—many fine texts are available for that purpose—but rather to introduce some basic concepts and terminology.

A computer network consists of machines interconnected by communication channels. We call these machines *hosts* and *routers*. Hosts are computers that run applications such as your Web browser, your IM agent, or a file-sharing program. The application programs running on hosts are the real “users” of the network. Routers are machines whose job is to relay, or *forward*, information from one communication channel to another. They may run programs but typically do not run application programs. For our purposes, a *communication channel* is a means of conveying sequences of bytes from one host to another; it may be a wired (e.g., Ethernet), a wireless (e.g., WiFi), or other connection.

Routers are important simply because it is not practical to connect every host directly to every other host. Instead, a few hosts connect to a router, which connects to other routers, and so on to form the network. This arrangement lets each machine get by with a relatively small number of communication channels; most hosts need only one. Programs that exchange information over the network, however, do not interact directly with routers and generally remain blissfully unaware of their existence.

By *information* we mean sequences of bytes that are constructed and interpreted by programs. In the context of computer networks, these byte sequences are generally called *packets*. A packet contains control information that the network uses to do its job and sometimes also includes user data. An example is information identifying the packet’s destination. Routers use such control information to figure out how to forward each packet.

A *protocol* is an agreement about the packets exchanged by communicating programs and what they mean. A protocol tells how packets are structured—for example, where the destination information is located in the packet and how big it is—as well as how the information is to be interpreted. A protocol is usually designed to solve a specific problem using given capabilities. For example, the *HyperTextTransferProtocol (HTTP)* solves the problem of transferring hypertext objects between servers, where they are stored or generated, and Web browsers that make them visible and useful to users. Instant messaging protocols solve the problem of enabling two or more users to exchange brief text messages.

Implementing a useful network requires solving a large number of different problems. To keep things manageable and modular, different protocols are designed to solve different sets of problems. TCP/IP is one such collection of solutions, sometimes called a *protocol suite*. It happens to be the suite of protocols used in the Internet, but it can be used in stand-alone private networks as well. Henceforth when we talk about the *network*, we mean any network that uses the TCP/IP protocol suite. The main protocols in the TCP/IP suite are the Internet Protocol (IP), the Transmission Control Protocol (TCP), and the User Datagram Protocol (UDP).

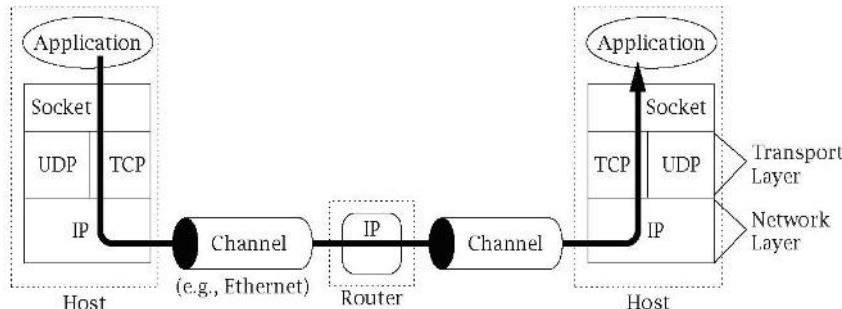


Fig 3.6 A TCP/IP network

It turns out to be useful to organize protocols into *layers*; TCP/IP and virtually all other protocol suites are organized this way. Figure 3.6 shows the relationships among the protocols, applications, and the sockets API in the hosts and routers, as well as the flow of data from one application (using TCP) to another. The boxes labeled TCP, UDP, and IP represent implementations of those protocols. Such implementations typically reside in the operating system of a host. Applications access the services provided by UDP and TCP through the sockets API. The arrow depicts the flow of data from the application, through the TCP and IP implementations, through the

network, and back up through the IP and TCP implementations at the other end.

In TCP/IP, the bottom layer consists of the underlying communication channels—for example, Ethernet or dial-up modem connections. Those channels are used by the *network layer*, which deals with the problem of forwarding packets toward their destination (i.e., what routers do). The single network layer protocol in the TCP/IP suite is the Internet Protocol; it solves the problem of making the sequence of channels and routers between any two hosts look like a single host-to-host channel.

The Internet Protocol provides a *datagram* service: every packet is handled and delivered by the network independently, like letters or parcels sent via the postal system. To make this work, each IP packet has to contain the *address* of its destination, just as every package that you mail is addressed to somebody. Although most delivery companies guarantee delivery of a package, IP is only a best-effort protocol: it attempts to deliver each packet, but it can (and occasionally does) lose, reorder, or duplicate packets in transit through the network.

The layer above IP is called the *transport layer*. It offers a choice between two protocols: TCP and UDP. Each builds on the service provided by IP, but they do so in different ways to provide different kinds of transport, which are used by *application protocols* with different needs. TCP and UDP have one function in common: addressing. Recall that IP delivers packets to hosts; clearly, a finer granularity of addressing is needed to get a packet to a particular application program, perhaps one of many using the network on the same host. Both TCP and UDP use addresses, called *port numbers*, to identify applications within hosts. TCP and UDP are called *end-to-end transport protocols* because they carry data all the way from one program to another (whereas IP only carries data from one host to another).

TCP is designed to detect and recover from the losses, duplications, and other errors that may occur in the host-to-host channel provided by IP. TCP provides a *reliable byte-stream* channel so that applications do not have to deal with these problems. It is a *connection-oriented* protocol: before using it to communicate, two programs must first establish a TCP connection, which involves completing an exchange of *hand shake messages* between the TCP implementations on the two communicating computers. Using TCP is also similar in many ways to file input/output (I/O). In fact, a file that is written by one program and read by another is a reasonable model of communication over a TCP connection. UDP, on the other hand, does not attempt to recover from errors experienced by IP; it simply extends the IP best-effort datagram service so that it works between application programs instead of between hosts. Thus, applications that use UDP must be prepared to deal with losses, reordering, and so on.

### 3.2.1.1 About Addresses

When you mail a letter, you provide the address of the recipient in a form that the postal service can understand. Before you can talk to someone on the phone, you must supply a phone number to the telephone system. In a similar way, before a program can communicate with another program, it must tell the network something to identify the other program. In TCP/IP, it takes two pieces of information to identify a particular program: an *Internet address*, used by IP, and a *port number*, the additional address interpreted by the transport protocol (TCP or UDP).

An *internet* (lower-case 'i') is a collection of computer networks that allows any computer on any of the associated networks to communicate with any other computer located on any of the other associated networks (or on the same network, of course). The protocol used for such communication is called the Internet Protocol (IP). *The Internet* (upper-case 'T') is the world's largest IP-based network.

Internet addresses are binary numbers. They come in two flavors, corresponding to the two versions of the Internet Protocol that have been standardized. The most common type is version 4 ("IPv4,"); the other is version 6 ("IPv6,"), which is just beginning to be deployed. IPv4 addresses are 32 bits long; because this is only enough to identify about 4 billion distinct destinations, they are not really big enough for today's Internet. For that reason, IPv6 was introduced. IPv6 addresses are 128 bits long. Many common Internet applications already work with IPv6 and it is expected that IPv6 will gradually replace IPv4, with the two coexisting for a number of years during a transition period.

Each computer on the Internet has a unique IP address, the current version of which is IPv4. This represents machine addresses in what is called quad notation. This is made up of four eight-bit numbers (i.e., numbers in the decimal range 0-255), separated by dots. For example, 131.122.3.219 would be one such address.

In writing down Internet addresses for human consumption (as opposed to using them inside programs), different

conventions are used for the two versions of IP. IPv4 addresses are conventionally written as a group of four decimal numbers separated by periods (e.g., 10.1.2.3); this is called the *dotted-quad* notation. The four numbers in a dotted-quad string represent the contents of the four bytes of the Internet address—thus, each is a number between 0 and 255.

The sixteen bytes of an IPv6 address, on the other hand, are represented as groups of hexadecimal digits, separated by colons (e.g., 2000:fdb8:0000:0000:0001:00ab:853c:39a1). Each group of digits represents two bytes of the address; leading zeros may be omitted, so the fifth and sixth groups in the foregoing example might be rendered as just :1:ab:. Also, consecutive groups that contain only zeros may be omitted altogether (but this can only be done once in any address). So the example above could be written as 2000:fdb8::1:00ab:853c:39a1.

Technically, each Internet address refers to the connection between a host and an underlying communication channel—in other words, a *network interface*. A host may have several interfaces; it is not uncommon, for example, for a host to have connections to both wired (Ethernet) and wireless (WiFi) networks. Because each such network connection belongs to a single host, an Internet address identifies a host as well as its connection to the network. However, the converse is not true, because a single host can have multiple interfaces, and each interface can have multiple addresses.

The *port number* in TCP or UDP is always interpreted relative to an Internet address. Returning to our earlier analogies, a port number corresponds to a room number at a given street address, say, that of a large building. The postal service uses the street address to get the letter to a mailbox; who ever empties the mailbox is then responsible for getting the letter to the proper room within the building. Or consider a company with an internal telephone system: to speak to an individual in the company, you first dial the company’s main phone number to connect to the internal telephone system and then dial the extension of the particular telephone of the individual you wish to speak with. In these analogies, the Internet address is the street address or the company’s main number, whereas the port corresponds to the room number or telephone extension. Port numbers are 16-bit unsigned binary numbers, so each one is in the range 1 to 65,535 (0 is reserved).

In each version of IP, certain special-purpose addresses are defined. One of these that is worth knowing is the *loopback address*, which is always assigned to a special *loopback interface*, a virtual device that simply echoes transmitted packets right back to the sender. The loopback interface is very useful for testing because packets sent to that address are immediately returned back to the destination. Moreover, it is present on every host, and can be used even when a computer has no other interfaces (i.e., is not connected to the network). The loopback address for IPv4 is 127.0.0.1; for IPv6 it is 0:0:0:0:0:0:1.

Another group of IPv4 addresses reserved for a special purpose includes those reserved for “private use.” This group includes all IPv4 addresses that start with 10 or 192.168, as well as those whose first number is 172 and whose second number is between 16 and 31. (There is no corresponding class for IPv6.) These addresses were originally designated for use in private networks that are *not* part of the global Internet. Today they are often used in homes and small offices that are connected to the Internet through a *network address translation (NAT)* device. Such a device acts like a router that translates (rewrites) the addresses and ports in packets as it forwards them. More precisely, it maps (private address, port) pairs in packets on one of its interfaces to (public address, port) pairs on the other interface. This enables a small group of hosts (e.g., those on a home network) to effectively “share” a single IP address. The importance of these addresses is that *they cannot be reached from the global Internet*.

A related class contains the *link-local*, or “autoconfiguration” addresses. For IPv4, such addresses begin with 169.254. For IPv6, any address whose first 16-bit chunk starts with FE8 is a link-local address. Such addresses can *only* be used for communication between hosts connected to the same network; routers will not forward them.

Finally, another class consists of *multicast* addresses. Whereas regular IP (sometimes called “unicast”) addresses refer to a single destination, multicast addresses potentially refer to an arbitrary number of destinations. In IPv4, multicast addresses in dotted-quad format have a first number in the range 224 to 239. In IPv6, multicast addresses start with FF.

### 3.2.1.2 About Names

Most likely you are accustomed to referring to hosts by *name* (e.g., host.example.com). However, the Internet protocols deal with addresses (binary numbers), not names. You should understand that the use of names instead

of addresses is a convenience feature that is independent of the basic service provided by TCP/IP—you can write and use TCP/IP applications without ever using a name. When you use a name to identify a communication endpoint, the system does some extra work to *resolve* the name into an address. This extra step is often worth it for a couple of reasons. First, names are obviously easier for humans to remember than dotted-quads (or, in the case of IPv6, strings of hexadecimal digits). Second, names provide a level of indirection, which insulates users from IP address changes. During the writing of the first edition of this book, the address of the Web server *www.mkp.com* changed. Because we always refer to that Web server by name, and because the change was quickly reflected in the service that maps names to addresses (about which we'll say more shortly)—*www.mkp.com* resolves to the current Internet address instead of 208.164.121.48—the change is transparent to programs that use the name to access the Web server.

The name-resolution service can access information from a wide variety of sources. Two of the primary sources are the *Domain Name System (DNS)* and local configuration databases. The DNS is a distributed database that maps *domain names* such as *www.mkp.com* to Internet addresses and other information; the DNS protocol allows hosts connected to the Internet to retrieve information from that database using TCP or UDP. Local configuration databases are generally OS-specific mechanisms for local name-to-Internet address mappings.

### 3.2.1.3 Clients, Servers and Peers

In a postal and telephone analogies, each communication is initiated by one party, who sends a letter or makes the telephone call, while the other party responds to the initiator's contact by sending a return letter or picking up the phone and talking. Internet communication is similar. The terms *client* and *server* refer to these roles: the client program initiates communication, while the server program waits passively for and then responds to clients that contact it.

Together, the client and server compose the *application*. The terms *client* and *server* are descriptive of the typical situation in which the server makes a particular capability—for example, a database service—available to any client that is able to communicate with it.

The most common categories of network software nowadays are *clients* and *servers*. These two categories have a symbiotic relationship and the term *client/server programming* has become very widely used in recent years. It is important to distinguish firstly between a server and the machine upon which the server is running (called the *host* machine), since I.T. workers often refer loosely to the host machine as 'the server'. Though this common usage has no detrimental practical effects for the majority of I.T. tasks, those I.T. personnel who are unaware of the distinction and subsequently undertake network programming are likely to be caused a significant amount of conceptual confusion until this distinction is made known to them.

A server, as the name implies, provides a service of some kind. This service is provided for clients that connect to the server's host machine specifically for the purpose of accessing the service. Thus, it is the clients that initiate a dialogue with the server. (These clients, of course, are also programs and are not human clients!) Common services provided by such servers include the 'serving up' of Web pages (by Web servers) and the downloading of files from servers' host machines via the File Transfer Protocol (FTP servers). For the former service, the corresponding client programs would be Web browsers (such as Netscape Communicator or Microsoft Explorer). Though a client and its corresponding server will normally run on different machines in a real-world application, it is perfectly possible for such programs to run on the *same* machine. Indeed, it is often very convenient (as will be seen in subsequent chapters) for server and client(s) to be run on the same machine, since this provides a very convenient 'sandbox' within which such applications may be tested before being released (or, more likely, before final testing on separate machines). This avoids the need for multiple machines and multiple testing personnel.

In some applications, such as messaging services, it is possible for programs on users' machines to communicate directly with each other in what is called *peer-to-peer* (or *P2P*) mode. However, for many applications, this is either not possible or prohibitively costly in terms of the number of simultaneous connections required. For example, the World Wide Web simply does not allow clients to communicate directly with each other. However, some applications use a server as an intermediary, in order to provide 'simulated' peer-to-peer facilities. Alternatively, both ends of the dialogue may act as both client and server. Peer-to-peer systems are beyond the intended scope of this text, though, and no further mention will be made of them.

Whether a program is acting as a client or server determines the general form of its use of the sockets API to establish communication with its *peer*. (The client is the peer of the server and vice versa.) Beyond that, the client-server distinction is important because the client needs to know the server's address and port initially, but not vice

versa. With the sockets API, the server can, if necessary, learn the client's address information when it receives the initial communication from the client. This is analogous to a telephone call—in order to be called, a person does not need to know the telephone number of the caller. As with a telephone call, once the connection is established, the distinction between server and client disappears.

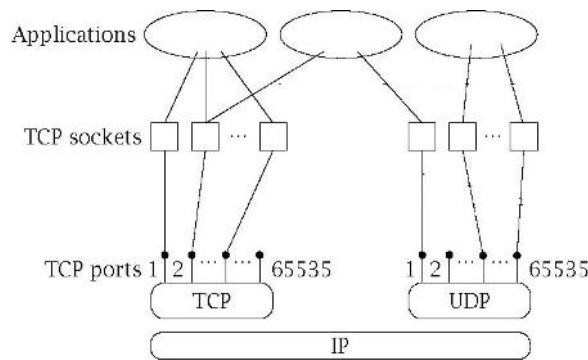
### 3.2.1.4 Sockets and Ports

These entities lie at the heart of network communications. For anybody not already familiar with the use of these terms in a network programming context, the two words very probably conjure up images of hardware components. However, although they are closely associated with the hardware communication links between computers within a network, *ports* and *sockets* are not themselves hardware elements, but abstract concepts that allow the programmer to make use of those communication links.

A *socket* is an abstraction through which an application may send and receive data, in much the same way as an open file handle allows an application to read and write data to stable storage. A socket allows an application to plug in to the network and communicate with other applications that are plugged in to the same network. Information written to the socket by an application on one machine can be read by an application on a different machine and vice versa.

A port is a *logical* connection to a computer (as opposed to a physical connection) and is identified by a number in the range 1-65535. This number has no correspondence with the number of physical connections to the computer, of which there may be only one (even though the number of ports used on that machine may be much greater than this). Ports are implemented upon all computers attached to a network, but it is only those machines that have server programs running on them for which the network programmer will refer explicitly to port numbers. Each port may be dedicated to a particular server/service (though the number of available ports will normally greatly exceed the number that is actually used). Port numbers in the range 1-1023 are normally set aside for the use of specified standard services, often referred to as 'well-known' services. For example, port 80 is normally used by Web servers. Some of the more common well-known services are listed in Section 1.4. Application programs wishing to use ports for non-standard services should avoid using port numbers 1-1023. (A range of 1024-65535 should be more than enough for even the most prolific of network programmers!)

Different types of sockets correspond to different underlying protocol suites and different stacks of protocols within a suite. This book deals only with the TCP/IP protocol suite. The main types of sockets in TCP/IP today are *stream sockets* and *datagram sockets*. Stream sockets use TCP as the end-to-end protocol (with IP underneath) and thus provide a reliable byte-stream service. A TCP/IP stream socket represents one end of a TCP connection. Datagram sockets use UDP (again, with IP underneath) and thus provide a best-effort datagram service that applications can use to send individual messages up to about 65,500 bytes in length. Stream and datagram sockets are also supported by other protocol suites, but this book deals only with TCP stream sockets and UDP datagram sockets. A TCP/IP socket is uniquely identified by an Internet address, an end-to-end protocol (TCP or UDP), and a port number. As you proceed, you will encounter several ways for a socket to become bound to an address.



**Fig. 3.7 Sockets, protocols and ports**

Figure 3.7 depicts the logical relationships among applications, socket abstractions, protocols, and port numbers within a single host. Note that a single socket abstraction can be referenced by multiple application programs. Each program that has a reference to a particular socket can communicate through that socket. Earlier we said that a port identifies an application on a host. Actually, a port identifies a socket on a host. From Figure 3.7, we see that multiple programs on a host can access the same socket. In practice, separate programs that access the same socket would usually belong to the same application (e.g., multiple copies of a Web server program), although in

principle they could belong to different applications.

For each port supplying a service, there is a server program waiting for any requests. All such programs run together in parallel on the host machine. When a client attempts to make connection with a particular server program, it supplies the port number of the associated service. The host machine examines the port number and passes the client's transmission to the appropriate server program for processing.

In most applications, of course, there are likely to be multiple clients wanting the same service at the same time. A common example of this requirement is that of multiple browsers (quite possibly thousands of them) wanting Web pages from the same server. The server, of course, needs some way of distinguishing between clients and keeping their dialogues separate from each other. This is achieved via the use of *sockets*. As stated earlier, a socket is an abstract concept and not an element of computer hardware. It is used to indicate one of the two end-points of a communication link between two processes. When a client wishes to make connection to a server, it will create a socket at its end of the communication link. Upon receiving the client's initial request (on a particular port number), the server will create a new socket at its end that will be dedicated to communication with that particular client. Just as one hardware link to a server may be associated with many ports, so too may one port be associated with many sockets.

### 3.2.1.5 Standard Services

In principle, servers can use any port, but the client must be able to learn what it is. In the Internet, there is a convention of assigning well-known port numbers to certain applications. The Internet Assigned Number Authority (IANA) oversees this assignment. For example, port number 21 has been assigned to the *File Transfer Protocol (FTP)*. When you run an FTP client application, it tries to contact the FTP server on that port by default. A list of all the assigned port numbers is maintained by the numbering authority of the Internet (see <http://www.iana.org/assignments/port-numbers>).

Whatever the service provided by a server, there must be some established *protocol* governing the communication that takes place between server and client. Each end of the dialogue must know what may/must be sent to the other, the format in which it should be sent, the sequence in which it must be sent (if sequence matters) and, for 'open-ended' dialogues, how the dialogue is to be terminated. For the standard services, such protocols are made available in public documents, usually by either the Internet Engineering Task Force (IETF) or the World Wide Web Consortium (W3C). Some of the more common services and their associated ports are shown in Table 3.1. For a more esoteric or 'bespoke' service, the application writer must establish a protocol and convey it to the intended users of that service.

**Table 3.1 Some well-known network services.**

Protocol name	Port number	Nature of service
Echo	7	The server simply echoes the data sent to it. This is useful for testing purposes.
Daytime	13	Provides the ASCII representation of the current date and time on the server.
FTP-data	20	Transferring files. (FTP uses two ports.)
FTP	21	Sending FTP commands like PUT and GET.
Telnet	23	Remote login and command line interaction.
SMTP	25	E-mail. (Simple Mail Transfer Protocol.)
HTTP	80	HyperText Transfer Protocol (the World Wide Web protocol).
NNTP	119	Usenet. (Network News Transfer Protocol.)

### 3.2.1.6 URLs and DNS

How does a client find out a server's IP address and port number? Usually, the client knows the name of the server it wants—for example, from a *Universal Resource Locator (URL)* such as <http://www.mkp.com>—and uses the name-resolution service to learn the corresponding Internet address.

A URL (Uniform Resource Locator) is a unique identifier for any resource located on the Internet. It has the following structure (in which BNF notation is used):

```
<protocol>://<hostname>[:<port>] [/<pathname>] [/<filename>[#<section>]]
```

For a well-known protocol, the port number may be omitted and the default port number will be assumed. Thus, since the example above specifies the HTTP protocol (the protocol of the Web) and does not specify on which port of the host machine the service is available, it will be assumed that the service is running on port 80 (the default port for Web servers). If the file name is omitted, then the server sends a default file from the directory specified in the path name. (This default file will commonly be called *index.html* or *default.html*.) The 'section' part of the URL (not often specified) indicates a named 'anchor' in an HTML document. For example, the HTML anchor in the tag

```
<A NAME="thisPlace"></A>
```

would be referred to as *thisPlace* by the section component of the URL.

Since human beings are generally much better at remembering meaningful strings of characters than they are at remembering long strings of numbers, the Domain Name System was developed. A *domain name*, also known as a *host name*, is the user-friendly equivalent of an IP address. In the previous example of a URL, the domain name was *java.sun.com*. The individual parts of a domain name don't correspond to the individual parts of an IP address. In fact, domain names don't always have four parts (as IPv4 addresses must have).

Normally, human beings will use domain names in preference to IP addresses, but they can just as well use the corresponding IP addresses (if they know what they are!). The *Domain Name System* provides a mapping between IP addresses and domain names and is held in a distributed database. The IP address system and the DNS are governed by ICANN (the Internet Corporation for Assigned Names and Numbers), which is a non-profitmaking organisation. When a URL is submitted to a browser, the DNS automatically converts the domain name part into its numeric IP equivalent.

### 3.2.2 Basic Sockets

Different processes (programs) can communicate with each other across networks by means of sockets. Java implements both TCP/IP sockets and datagram sockets (UDP sockets). Very often, the two communicating processes will have a *client/server* relationship. The steps required to create client/server programs via each of these methods are very similar and are outlined in the following sections.

The `java.net` package provides an object-oriented framework for the creation and use of Internet Protocol (IP) sockets. In this section, we'll take a look at these classes and what they offer.

Before communicating with another party, you must first know how to address your messages so they can be delivered correctly. Notice that I didn't say that you need to know where the other party is located—once a scheme for encoding a location is established, I simply need to know my party's encoded address to communicate. On IP networks, the addressing scheme in use is based on hosts and port numbers.

A given host computer on an IP network has a hostname and a numeric address. Either of these, in their fully qualified forms, is a unique identifier for a host on the network. The JavaSoft home page, for example, resides on a host named *www.javasoft.com*, which currently has the IP address *204.160.241.98*. Either of these addresses can be used to locate the machine on an IP network. The textual name for the machine is called its Domain Name Services (DNS) name, which can be thought of as a kind of alias for the numeric IP address.

In the Java API, the `InetAddress` class represents an IP address. You can query an `InetAddress` for the name of the host using its `getHostName()` method, and for its numeric address using `getAddress()`. Notice that, even though we can uniquely specify a host with its IP address, we do not necessarily know its physical location. I look at the web pages on *www.javasoft.com* regularly, but I don't know where the machine is. Conversely, even if I knew where

the machine was physically, it wouldn't do me a bit of good if I didn't know its IP address (unless someone was kind enough to label the machine with it, or left a terminal window open on the server's console for me to get its IP address directly).

Now, you typically don't want to communicate with a given host, but rather with one or many agent processes running on the host. To engage in network communications, each process must associate itself with a port on the host, identified by a number. HTTP servers, for example, typically attach themselves to port 80 on their host machine. When you ask to connect to <http://www.javasoft.com/> from your web browser, the browser automatically assumes the default port and attempts to connect to the process running on [www.javasoft.com](http://www.javasoft.com) listening to port 80. If this process is an HTTP server process that understands the commands that the browser is sending, the browser and the server will commence communications.

This host/port scheme is the basis of the IP addressing protocol, and is supported directly in the Java API. All network connections are specified using an Inet–Address and a port number. The Java environment does the hard work of initiating the IP protocol communications and creating Java objects that represent these network connections.

### 3.2.2.1 Socket Addresses

Recall that a client must specify the IP address of the host running the server program when it initiates communication. The network infrastructure then uses this *destination address* to route the client's information to the proper machine. Addresses can be specified in Java using a string that contains either a numeric address—in the appropriate form for the version, e.g., 192.0.2.27 for IPv4 or fe20:12a0::0abc:1234 for IPv6—or a name (e.g., *server.example.com*). In the latter case the name must be *resolved* to a numerical address before it can be used for communication.

The InetAddress abstraction represents a network destination, encapsulating both names and numerical address information. The class has two subclasses, Inet4Address and Inet6Address, representing the two versions in use. Instances of InetAddress are immutable: once created, each one always refers to the same address. We'll demonstrate the use of InetAddress with an example program that first prints out all the addresses—IPv4 and IPv6, if any—associated with the local host, and then prints the names and addresses associated with each host specified on the command line.

To get the addresses of the local host, the program takes advantage of the NetworkInterface abstraction. Recall that IP addresses are actually assigned to the connection between a host and a network (and not to the host itself). The NetworkInterface class provides access to information about all of a host's interfaces. This is extremely useful, for example when a program needs to inform another program of its address.

#### InetAddressExample.java

```
0 import java.util.Enumeration;
1 import java.net.*;
2
3 public class InetAddressExample{
4
5 public static void main(String[]args) {
6
7 //Get the network interface and associated addresses for this host
8 try{
9     Enumeration <NetworkInterface> interfaceList=NetworkInterface.getNetworkInterfaces();
10    if(interfaceList==null){
11        System.out.println("--No interfaces found--");
12    } else {
13        while(interfaceList.hasMoreElements()){
14            NetworkInterface iface=interfaceList.nextElement();
15            System.out.println("Interface"+iface.getName()+":");
16            Enumeration <InetAddress> addrList=iface.getInetAddresses();
17            if(!addrList.hasMoreElements()){
18                System.out.println("\tNo addresses for this interface");
19            }
20            while(addrList.hasMoreElements()){
21                InetAddress address=addrList.nextElement();
22                System.out.print("\tAddress"

```

```

23         + ((address instanceof Inet4Address ? "(v4)"
24             :(address instanceof Inet6Address?"(v6)":"(?))))));
25         System.out.println(": "+address.getHostAddress());
26     }
27 }
28 }catch(SocketException e){
29     System.out.println("Error getting network interfaces:"+e.getMessage());
30 }
31 }
32
33 //Get name(s)/address(es) of hosts give non command line
34 for(String host:args{
35     try{
36         System.out.println(host+":");
37         InetAddress[] addressList=InetAddress.getAllByName(host);
38         for(InetAddress address:addressList){
39             System.out.println("\t"+address.getHostName() +"/"+address.getHostAddress());
40         }
41     }catch(UnknownHostException e){
42         System.out.println("\tUnable to find address for "+host);
43     }
44 }
45 }
46 }

```

1       Get a list of this host's network interfaces: line 9. The static method `getNetworkInterfaces()` returns a list containing an instance of `NetworkInterface` for each of the host's interfaces.

2       Check for empty list: lines 10–12. The loopback interface is generally always included, even if the host has no other network connection, so this check will succeed only if the host has no networking subsystem at all.

3.      Get and print address(es) of each interface in the list: lines 13–27

- Print the interface's name: line 15 The `getName()` method returns a local name for the interface. This is usually a combination of letters and numbers indicating the type and particular instance of the interface—for example, “`lo0`” or “`eth0`”.
- Get the addresses associated with the interface: line 16 The `getInetAddresses()` method returns another `Enumeration`, this time containing instances of `InetAddress`—one per address associated with the interface. Depending on how the host is configured, the list may contain only IPv4, only IPv6, or a mixture of both types of address.
- Check for empty list: lines 17–19
- Iterate through the list, printing each address: lines 20–26
- We check each instance to determine which subtype it is.(At this time the only subtypes of `InetAddress` are those listed, but conceivably there might be others someday.) The `getHostAddress()` method of `InetAddress` returns a String representing the numerical address in the format appropriate for its specific type: dotted-quad for v4, colon-separated hex for v6.

4      Catch exception: lines 29–31.The call to `getNetworkInterfaces()` can throw a `SocketException`.

5      Get names and addresses for each command-line argument: lines 34–44

- Get list of addresses for the given name/address: line 37
- Iterate through the list, printing each: lines 38–40 For each host in the list, we print the name returned by `getHostName()` followed by the numerical address returned by `getHostAddress()`.

To use this application to find information about the local host, the publisher's Web server ([www.mkp.com](http://www.mkp.com)), a fake name (`blah.blah`), and an IP address, do the following:

```
% java InetAddressExample www.mkp.com blah.blah 129.35.69.7
Interface lo:
Address (v4): 127.0.0.1
Address (v6): 0:0:0:0:0:0:0:1
Address (v6): fe80:0:0:0:0:0:0:1%1
Interface eth0:
Address (v4): 192.168.159.1
Address (v6): fe80:0:0:250:56ff:fed0:8%4
www.mkp.com:
www.mkp.com/129.35.69.7
blah.blah:
Unable to find address for blah.blah
129.35.69.7:
129.35.69.7/129.35.69.7
```

You may notice that some v6 addresses have a suffix of the form `%d`, where  $d$  is a number. Such addresses have limited scope (typically they are link-local), and the suffix identifies the particular scope with which they are associated; this ensures that each listed address string is unique. Link-local IPv6 addresses begin with fe8.

You may also have noticed a delay when resolving blah.blah. Your resolver looks in several places before giving up on resolving a name. When the name service is not available for some reason—say, the program is running on a machine that is not connected to any network—attempting to identify a host by name may fail. Moreover, it may take a significant amount of time to do so, as the system tries various ways to resolve the name to an IP address. It is, therefore, good to know that you can always refer to a host using the IP address in dotted-quad notation. In any of the examples in this book, if a remote host is specified by name, the host running the example must be configured to convert names to addresses, or the example won’t work. If you can ping a host using one of its names (e.g., run the command “`ping server.example.com`”), then the examples should work with names. If your ping test fails or the example hangs, try specifying the host by IP address, which avoids the name-to-address conversion altogether.

#### *InetAddress: Creating and accessing*

```
Static InetAddress[] getAllByName(String host)
static InetAddress getByName(String host)
static InetAddress getLocalHost()
byte[] getAddress()
```

The static factory methods return instances that can be passed to other Socket methods to specify a host. The input String to the factory method scan be either a domain name, such as “skeezix” or “farm.example.com”, or a string representation of a numeric address. A name may be associated with more than one numeric address; the `getAllByName()` method returns an instance for each address associated with a name. The `getAddress()` method returns the binary form of the address as a byte array of appropriate length. If the instance is of `Inet4Address`, the array is four bytes in length; if of `Inet6Address`, it is 16 bytes. The first element of the returned array is the most significant byte of the address.

As we have seen, an `InetAddress` instance may be converted to a String representation in several ways.

#### *InetAddress: String representations*

```
String toString() String getHostAddress()
String getHostName() String getCanonicalHostName()
```

These methods return the name or numeric address of the host, or a combi-nation thereof, as a properly formatted String. The `toString()` method overrides the `Object` method to return a string of the form “`hostname.example.com/192.0.2.127`” or “`never.example.net/2000::620:1a30:95b2`”. The numeric representation of the address (only) is returned by `getHostAddress()`. For an IPv6 address, the string representation always includes the full eight groups (i.e., exactly seven colons “`:`”) to prevent ambiguity when a port number is appended separated by another colon—a common idiom that we’ll see later. Also, an IPv6 address that has limited scope, such as a link-local address will have a *scope identifier* appended. This is a local identifier added to prevent ambiguity (since the same link-local address can be used on different links), but is not part of the address transmitted in the packet. The last two methods return the name of the host only, their behavior differing as follows: If this instance was originally created by giving a name, `getHostName()` will return that name with no resolution step; otherwise, `getHostName()` resolves the address to the name using the system-configured resolution mechanism. The `getCanonicalName()` method, on the other hand, always tries to resolve the address to obtain a *fully qualified domain name* (like “`ns1.internat.net`” or “`bam.example.com`”). Note that that address might differ from the one with which the instance was created, if different names map to the same address. Both meth-ods return the numerical form of the address if resolution cannot be completed. Also, both check permission with the security manager before sending any messages. The `InetAddress` class also supports checking for properties, such as membership in a class of “special purpose” addresses as discussed in Section 1.2, and reachability, i.e., the ability to exchange packets with the host.

#### *InetAddress: Testing properties*

```
boolean isAnyLocalAddress() boolean isLinkLocalAddress() boolean isLoopbackAddress()
boolean isMulticastAddress() boolean isMCGlobal() boolean isMCLinkLocal()
boolean isMCNodeLocal() boolean isMCOrgLocal() boolean isMCSiteLocal()
boolean isReachable(int timeout)
boolean isReachable(NetworkInterface netif,int ttl,int timeout)
```

These methods check whether an address is of a particular type. They all work for both IPv4 and IPv6 addresses. The first three methods above check whether the instance is one of, respectively, the “don’t care” address, an address in the link-local class, or the loopback address (matches `127.*.*.*` or `::1`). The fourth method checks whether it is a multicast address (see Section 4.3.2), and the `isMC...()` methods check for various *scopes* of multicast address. (The scope determines, roughly, how far packets addressed to that destination can travel from their origin.) The last two methods check whether it is actually possible to exchange packets with the host identified by this `InetAddress`. Note that, unlike the other methods, which involve simple syntactic checks, these methods cause the networking system to take action, namely sending packets. The system attempts to send a packet until the specified number of milliseconds passes. The latter form is more specific: it determines whether the destination can be contacted by sending packets out over the specified `NetworkInterface`, with the specified *time-to-*

*live (TTL)* value. The TTL limits the distance a packet can travel through the network. Effectiveness of these last two methods may be limited by the security manager configuration. The NetworkInterface class provides a large number of methods, many of which are beyond the scope of this book. We describe here the most useful ones for our purposes.

#### *NetworkInterface: Creating, getting information*

```
Static Enumeration<NetworkInterface> getNetworkInterfaces()
static NetworkInterface getByInetAddress(InetAddress addr)
static NetworkInterface getByName(String name)
Enumeration<InetAddress> getInetAddresses()
String getName()
String getDisplayName()
```

The first method above is quite useful, making it easy to learn an IP address of the host a program is running on: you get the list of interfaces with `getNetworkInterfaces()`, and use the `getInetAddresses()` instance method to get all the addresses of each. The list contains *all* the interfaces of the host, including the loopback virtual interface, which cannot send or receive messages to the rest of the network. Similarly, the list of addresses may contain link-local addresses that also are not globally reachable. Since the order is unspecified, you cannot simply take the first address of the first interface and assume it can be reached from the Internet; instead, use the property-checking methods of `InetAddress`(see above) to find one that is not loopback, not link-local, etc.

The `getName()` methods return the name of the *interface* (not the host). This generally consists of an alphabetic string followed by a numeric part, for example `eth0`. The loopback interface is named `lo0` on many systems.

#### Second Example

Static method `getByName` of the `InetAddress` uses DNS (Domain Name System) to return the Internet address of a specified host name as an `InetAddress` object. In order to display the IP address from this object, we can simply use method `println` (which will cause the object's `toString` method to be executed). Since method `getByName` throws the checked exception `UnknownHostException` if the host name is not recognised, we must either throw this exception or (preferably) handle it with a catch clause. The following example illustrates this.

```
import java.net.*;
import java.util.*;
public class IPFinder
{
    public static void main(String[] args) {
        String host;
        Scanner input = new Scanner(System.in);
        System.out.print("\n\nEnter host name: ");
        host = input.next();
        try
        { InetAddress address = InetAddress.getByName(host);
            System.out.println("IP address: " + address.toString());
        }
        catch (UnknownHostException uhEx)
        { System.out.println("Could not find " + host); } } }
```

It is sometimes useful for Java programs to be able to retrieve the IP address of the current machine. The example below shows how to do this.

#### Third Example

```
import java.net.*;
public class MyLocalIPAddress
{
    public static void main(String[] args) {
        try
        {
            InetAddress address = InetAddress.getLocalHost();
            System.out.println(address);
        }
        catch (UnknownHostException uhEx) {
            System.out.println( "Could not find local address!"); } } }
```

### 3.2.2.2 Fast overview of Java classes for TCP and UDP sockets

At the core of Java's networking support are the `Socket` and `DatagramSocket` classes in `java.net`. These classes define channels for communication between processes over an IP network. A new socket is created by specifying a host, either by name or with an `InetAddress` object, and a port number on the host. There are two basic flavors of network sockets on IP networks: those that use the Transmission Control Protocol (TCP) and those that use the User Datagram Protocol (UDP). TCP is a reliable protocol in which data packets are guaranteed to be delivered, and delivered in order. If a packet expected at the receiving end of a TCP socket doesn't arrive in a set period of time, then it is assumed lost, and the packet is requested from the sender again. The receiver doesn't move on to the next packet until the first is received. UDP, on the other hand, makes no guarantees about delivery of packets, or the order in which packets are delivered. The sender transmits a UDP packet, and it either makes it to the receiver or it doesn't. TCP sockets are used in the large majority of IP applications. UDP sockets are typically used in bandwidth-limited applications, where the overhead associated with resending packets is not tolerable. A good example of this is real-time network audio applications. If you are delivering packets of audio information over the network to be played in real-time, then there is no point in resending a late packet. By the time it gets delivered it will be useless, since the audio track must play continuously and sequentially, without backtracking.

The `Socket` class is used for creating TCP connections over an IP network. A `Socket` is typically created using an `InetAddress` to specify the remote host, and a port number to which the host can connect. A process on the remote host must be listening on that port number for incoming connection requests. In Java, this can be done using a `ServerSocket`:

```
// Listen to port 5000 on the local host for socket connection requests
ServerSocket s = new ServerSocket(5000);
while (true) {
    // Wait for a connection request from a client
    Socket clientConn = s.accept();
    InputStream in = clientConn.getInputStream();
    OutputStream out = clientConn.getOutputStream();
    // Now we have input and output streams connected to our client, do
    // something with them...
```

On client side, the code simply creates a socket to the remote host on the specified port (5000, in this case):

```
// Create the socket
InetAddress addr = InetAddress.getByName("our.remote.host");
Socket s = new Socket(addr, 5000);
InputStream in = s.getInputStream();
OutputStream out = s.getOutputStream();
// We've got input/output streams to the remote process,
// now do something with them...
```

UDP socket connections are created and used through the `DatagramSocket` and `DatagramPacket` classes. A `DatagramSocket` sends and receives data using UDP packets, represented as `DatagramPacket` objects. Before two agents can talk to each other over a UDP connection, they both have to have a `DatagramSocket` connected to a port on their local machines. This is done by simply creating a `DatagramSocket` object:

```
DatagramSocket udpSocket = new DatagramSocket(5000);
```

In this example we are connecting a UDP socket to a specific port (5000) on the local host. If we don't particularly care which port is used, then we can construct the `DatagramSocket` without specifying the port. An unused port on the local host will be used, and we can find out which one by asking the new socket for its port number:

```
DatagramSocket udpSocket = new DatagramSocket();
int portNo = udpSocket.getLocalPort();
```

In order for two agents to send data to each other over a UDP socket, they must know the host name and port number of each other's socket connection. So they will either have preordained ports for each other and will create `DatagramSocket` using these port numbers, or they will create a socket on a random local port and transmit their port numbers to each other over another connection.

Data is sent over a DatagramSocket using DatagramPacket objects. Each DatagramPacket contains a data buffer, the address of the remote host to send the data to, and the port number the remote agent is listening to. So to send a buffer of data to a process listening to port 5000 on host *my.host.com*, we would do something like this:

```
byte[] dataBuf = {'h', 'i', ' ', 't', 'h', 'e', 'r', 'e'};  
InetAddress addr = InetAddress.getByName("my.host.com");  
DatagramPacket p = new DatagramPacket(dataBuf, dataBuf.length, addr, 5000);  
udpSocket.send(p);
```

The remote process can receive the data in the form of a DatagramPacket by calling the receive() method on its DatagramSocket. The received DatagramPacket will have the host address and port of the sender filled in as a side-effect of the call.

Note that in all of the examples, we would have to catch the appropriate exceptions and handle them. Sending a DatagramPacket, for example, can generate an IOException if the network transmission fails for some reason. A robust networked program will catch this exception and behave appropriately, perhaps by resending the packet if the application warrants, or perhaps by simply noting the lost packet and continuing.

### 3.2.2.3 Multicast Sockets

There is a subset of the IP protocol that supports *multicasting*. Multicasting can be thought of as broadcasting data over a network connection to many connected agents, as opposed to unicasting packets between two agents on a normal connection. Multicasting is done using UDP packets that are broadcast out on a multicast IP address. Any agent "listening in" to that IP address will receive the data packets that are broadcast. The analogy to radio and television broadcasting is no accident—the very first practical uses of multicast IP were for broadcasting audio and video over the Internet from special events.

Java supports multicast IP through the `java.net.MulticastSocket` class, which is an extension of the `DatagramSocket` class. Joining a multicast group is done almost the same way that you would establish a UDP connection between two agents. Each agent that wants to listen on the multicast address creates a `MulticastSocket` and then joins the multicast session by calling the `joinGroup()` method on the `MulticastSocket`:

```
MulticastSocket ms = new MulticastSocket();  
InetAddress sessAddr = InetAddress.getByName("224.2.76.24");  
ms.joinGroup(sessAddr);
```

Once the connection to the multicast session is established, the agent can read data being broadcast on the multicast "channel":

```
byte[] audioBuf = new byte[1024];  
DatagramPacket dp = new DatagramPacket(audioBuf, 1024);  
ms.receive(dp);  
// Play the data on a fictitious audio device  
myAudioDevice.play(dp.getData());
```

Data can also be sent out on the multicast channel to all the other listening agents using the `send()` method on the `MulticastSocket`. Once the broadcast is over, or we simply want to stop listening, we can disconnect from the session using the `leaveGroup()` method:

```
ms.leaveGroup(sessAddr);
```

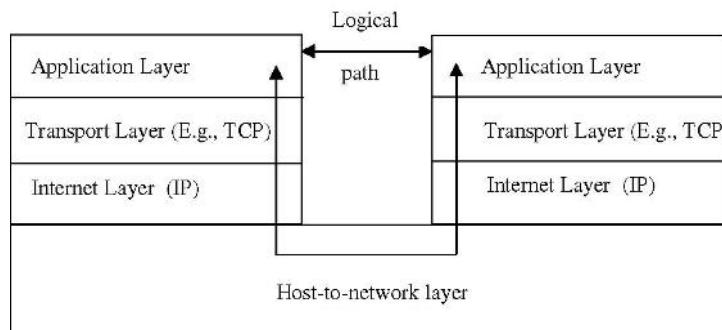
Multicasting is useful when we want to connect many agents together on a common communication channel. Shared audio and video channels are the most obvious uses, but multicasting can also be applied in collaborative tools like shared whiteboards, or between application servers performing synchronization tasks, like load balancing. However, since multicast IP is based on UDP, you have to be willing to accept the possibility of losing some data along the way, and dealing with it gracefully. Also, since clients can join a multicast session asynchronously, they have to be ready to synchronize themselves with the current state of the multicast session when they join.

### 3.2.3 TCP Sockets

In common with all modern computer networks, the Internet is a packet-switched network, which means that messages between computers on the Internet are broken up into blocks of information called packets, with each packet being handled separately and possibly travelling by a completely different route from that of other such packets from the same message. IP is concerned with the routing of these packets through an internet. Introduced by the American military during the Cold War, it was designed from the outset to be robust. In the event of a military strike against one of the network routers, the rest of the network had to continue to function as normal, with messages that would have gone through the damaged router being re-routed. IP is responsible for this re-routing. It attaches the IP address of the intended recipient to each packet and then tries to determine the most efficient route available to get to the ultimate destination (taking damaged routers into account).

However, since packets could still arrive out of sequence, be corrupted or even not arrive at all (without indication to either sender or intended recipient that anything had gone wrong), it was decided to place another protocol layer on top of IP. This further layer was provided by TCP (Transmission Control Protocol), which allowed each end of a connection to acknowledge receipt of IP packets and/or request retransmission of lost or corrupted packets. In addition, TCP allows the packets to be rearranged into their correct sequence at the receiving end. IP and TCP are the two commonest protocols used on the Internet and are almost invariably coupled together as TCP/IP. TCP is the higher level protocol that uses the lower level IP.

For Internet applications, a four-layer model is often used, which is represented diagrammatically in Figure 3.8. The transport layer will often comprise the TCP protocol, but may be UDP (described in the next section), while the internet layer will always be IP. Each layer of the model represents a different level of abstraction, with higher levels representing higher abstraction. Thus, although applications may appear to be communicating directly with each other, they are actually communicating directly only with their transport layers. The transport and internet layers, in their turn, communicate directly only with the layers immediately above and below them, while the host-to-network layer communicates directly only with the IP layer at each end of the connection. When a message is sent by the application layer at one end of the connection, it passes through each of the lower layers. As it does so, each layer adds further protocol data specific to the particular protocol at that level. For the TCP layer, this process involves breaking up the data packets into TCP segments and adding sequence numbers and checksums; for the IP layer, it involves placing the TCP segments into IP packets called datagrams and adding the routing details. The host-to-network layer then converts the digital data into an analogue form suitable for transmission over the carrier wire, sends the data and converts it back into digital form at the receiving end.



**Fig. 3.8 The 4-layer Network Model**

At the receiving end, the message travels up through the layers until it reaches the receiving application layer. As it does so, each layer converts the message into a form suitable for receipt by the next layer (effectively reversing the corresponding process carried out at the sending end) and carries out checks appropriate to its own protocol. If recalculation of checksums reveals that some of the data has been corrupted or checking of sequence numbers shows that some data has not been received, then the transport layer requests re-transmission of the corrupt/missing data. Otherwise, the transport layer acknowledges receipt of the packets. All of this is completely transparent to the application layer. Once all the data has been received, converted and correctly sequenced, it is presented to the recipient application layer as though that layer had been in direct communication with the sending application layer. The latter may then send a response in exactly the same manner (and so on). In fact, since TCP provides full duplex transmission, the two ends of the connection may be sending data simultaneously.

The above description has deliberately hidden many of the low-level details of implementation, particularly the tasks carried out by the host-to-network layer. In addition, of course, the initial transmission may have passed

through several routers and their associated layers before arriving at its ultimate destination. However, this high-level view covers the basic stages that are involved and is quite sufficient for our purposes.

Another network model that is often referred to is the seven-layer Open Systems Interconnection (OSI) model. However, this model is an unnecessarily complex one for our purposes and is better suited to non-TCP/IP networks anyway.

### 3.2.3.1 TCP Sockets in Java

Java provides two classes for TCP: `Socket` and `ServerSocket`. An instance of `Socket` represents one end of a TCP connection. A *TCP connection* is an abstract two-way channel whose ends are each identified by an IP address and port number. Before being used for communication, a TCP connection must go through a setup phase, which starts with the client's TCP sending a connection request to the server's TCP. An instance of `ServerSocket` listens for TCP connection requests and creates a new `Socket` instance to handle each incoming connection. Thus, servers handle both `ServerSocket` and `Socket` instances, while clients use only `Socket`. We begin by examining an example of a simple client.

### 3.2.3.2 Example of TCP Client

The client initiates communication with a server that is passively waiting to be contacted. The typical TCP client goes through three steps:

- 1 Construct an instance of `Socket`: The constructor establishes a TCP connection to the specified remote host and port.
- 2 Communicate using the socket's I/O streams: A connected instance of `Socket` contains an `InputStream` and `OutputStream` that can be used just like any other Java I/O stream.
- 3 Close the connection using the `close()` method of `Socket`.

Our first TCP application, called `TCPEchoClient.java`, is a client that communicates with an *echo server* using TCP. An echo server simply repeats whatever it receives back to the client. The string to be echoed is provided as a command-line argument to our client. Some systems include an echo server for debugging and testing purposes. You may be able to use a program such as telnet to test if the standard echo server is running on your system.

#### ***TCPEchoClient.java***

```
0 import java.net.Socket;
1 import java.net.SocketException;
2 import java.io.IOException;
3 import java.io.InputStream;
4 import java.io.OutputStream;
5
6 public class TCPEchoClient{
7
8 public static void main(String[] args) throws IOException{
9
10 if((args.length<2) || (args.length>3))//Test for correct # of args
11 throw new IllegalArgumentException("Parameter(s) :<Server><Word>[<Port>]");
12
13 String server=args[0];//Server name or IP address
14 //Convert argument String to bytes using the default character encoding
15 byte[] data=args[1].getBytes();
16
17 int servPort=(args.length==3)?Integer.parseInt(args[2]):7;
18
19 //Create socket that is connected to server on specified port
20 Socket socket=new Socket(server,servPort);
21 System.out.println("Connected to server...sending echo string");
22
23 InputStream in=socket.getInputStream();
24 OutputStream out=socket.getOutputStream();
25
26 out.write(data);//Send the encoded string to the server
27
28 //Receive the same string back from the server
29 int totalBytesRcvd=0; //Total bytes received so far
```

```

30 int bytesRcvd; //Bytes received in last read
31 while(totalBytesRcvd<data.length){
32     if((bytesRcvd=in.read(data,totalBytesRcvd,
33         data.length-totalBytesRcvd))==-1)
34         throw new SocketException("Connection closed prematurely");
35     totalBytesRcvd+=bytesRcvd;
36 } //data array is full
37
38 System.out.println("Received: "+newString(data));
39
40 socket.close(); //Close the socket and its streams
41 }
42 }

```

1. Application setup and parameter parsing: lines 0–17

- Convert the echo string: line 15 TCP sockets send and receive sequences of bytes. The getBytes() method of String returns a byte array representation of the string.
- Determine the port of the echo server: line 17. The default echo port is 7. If we specify a third parameter, Integer.parseInt() takes the string and returns the equivalent integer value.

2. TCP socket creation: line 20. The Socket constructor creates a socket and connects it to the specified server, identified either by name or IP address, before returning. Note that the underlying TCP deals only with IP addresses; if a name is given, the implementation resolves it to the corresponding address. If the connection attempt fails for any reason, the constructor throws an IOException.

3. Get socket input and output streams: lines 23–24. Associated with each connected Socket instance is an InputStream and an OutputStream. We send data over the socket by writing bytes to the OutputStream just as we would any other stream, and we receive by reading from the InputStream.

4. Send the string to echo server: line 26. The write() method of OutputStream transmits the given byte array over the connection to the server.

5. Receive the reply from the echo server: lines 29–36. Since we know the number of bytes to expect from the echo server, we can repeatedly receive bytes until we have received the same number of bytes we sent. This particular form of read() takes three parameters: 1) byte array to receive into, 2) byte offset into the array where the first byte received should be placed, and 3) the maximum number of bytes to be placed in the array. read() blocks until some data is available, reads up to the specified maximum number of bytes, and returns the number of bytes actually placed in the array (which may be less than the given maximum). The loop simply fills up *data* until we receive as many bytes as we sent. If the TCP connection is closed by the other end, read() returns -1. For the client, this indicates that the server prematurely closed the socket.

Why not just a single read? TCP does not preserve read() and write() message boundaries. That is, even though we sent the echo string with a single write(), the echo server may receive it in multiple chunks. Even if the echo string is handled in one chunk by the echo server, the reply may still be broken into pieces by TCP. One of the most common errors for beginners is the assumption that data sent by a single write() will always be received in a single read().

6. Print echoed string: line 38. To print the server's response, we must convert the byte array to a string using the default character encoding.

7. Close socket: line 40. When the client has finished receiving all of the echoed data, it closes the socket.

We can communicate with an echo server named *server.example.com* with IP address 192.0.2.1 in either of the following ways:

```
%java TCPEchoClient server.example.com "Echo this!"Received:Echothis!%java
TCPEchoClient 192.0.2.1 "Echo this!"

Received: Echo this!
```

*Socket:Creation*

```

Socket(InetAddress remoteAddr, int remotePort)
Socket(String remoteHost, int remotePort)
Socket(InetAddress remoteAddr, int remotePort, InetAddress localAddr, int localPort)
Socket(String remoteHost, int remotePort, InetAddress localAddr, int localPort)
Socket()

```

The first four constructors create a TCP socket and *connect* it to the specified remote address and port before returning. The first two do not specify the local address and port, so a default local address and some available port are chosen. Specifying the local address may be useful on a host with multiple interfaces. String arguments that specify destinations can be in the same formats that are accepted by the InetAddress creation methods. The last constructor creates an unconnected socket, which must be explicitly connected (via the connect() method, see below) before it can be used for communication.

*Socket:Operations*

```

void connect(SocketAddress destination)
void connect(SocketAddress destination, int timeout)
InputStream getInputStream()
OutputStream getOutputStream()
void close()
```

```

void shutdownInput()
void shutdownOutput()

```

The connect() methods cause a TCP connection to the specified endpoints to be opened. The abstract class `SocketAddress` represents a generic form of address for a socket; its subclass `InetSocketAddress` is specific to TCP/IP sockets (see description below). Communication with the remote system takes place via the associated I/O streams, which are obtained through the `get...Stream()` methods. The `close()` method closes the socket and its associated I/O streams, preventing further operations on them. The `shutDownInput()` method closes the input side of a TCP stream. Any unread data is silently discarded, including data buffered by the socket, data in transit, and data arriving in the future. Any subsequent attempt to read from the socket will cause an exception to be thrown. The `shutDownOutput()` method has a similar effect on the output stream, but the implementation will attempt to ensure that any data already written to the socket's output stream is delivered to the other end. By default, `Socket` is implemented on top of a TCP connection; however, in Java, you can actually change the underlying implementation of `Socket`. This book is about TCP/IP, so for simplicity we assume that the underlying implementation for all of these networking classes is the default.

#### *Socket: Getting/testing attributes*

```

InetAddress getInetAddress()
int getPort()
InetAddress getLocalAddress()
int getLocalPort()
SocketAddress getRemoteSocketAddress()
SocketAddress getLocalSocketAddress()

```

These methods return the indicated attributes of the socket, and any method in this book that returns a `SocketAddress` actually returns an instance of `InetSocketAddress`. The `InetSocketAddress` encapsulates an `InetAddress` and a port number. The `Socket` class actually has a large number of other associated attributes referred to as *socket options*.

#### *InetSocketAddress: Creating and accessing*

```

InetSocketAddress(InetAddress addr, int port)
InetSocketAddress(int port)
InetSocketAddress(String hostname, int port)
static InetSocketAddress createUnresolved(String host, int port)
boolean isUnresolved()
InetAddress getAddress()
int getPort()
String getHostName()
String toString()

```

The `InetSocketAddress` class provides an immutable combination of host address and port. The port-only constructor uses the special “any” address, and is useful for servers. The constructor that takes a string hostname attempts to resolve the name to an IP address; the static `createUnresolved()` method allows an instance to be created without attempting this resolution step. The `isUnresolved()` method returns true if the instance was created this way, or if the resolution attempt in the constructor failed. The `get...()` methods provide access to the indicated components, with `getHostName()` providing the name associated with the contained `InetAddress`. The `toString()` method overrides that of `Object` and returns a string consisting of the name associated with the contained address (if known), a ‘/’ (slash), the address in numeric form, a ‘:’ (colon), and the port number. If the `InetSocketAddress` is unresolved, only the String with which it was created precedes the colon.

### 3.2.3.3 Example of TCP Server

We now turn our attention to constructing a server. The server’s job is to set up a communication endpoint and passively wait for connections from clients. The typical TCP server goes through two steps:

- 1 Construct a `ServerSocket` instance, specifying the local port. This socket listens for incoming connections to the specified port.
2. Repeatedly:
  - a. Call the `accept()` method of `ServerSocket` to get the next incoming client connection. Upon establishment of a new client connection, an instance of `Socket` for the new connection is created and returned by `accept()`.
  - b. Communicate with the client using the returned `Socket`’s `InputStream` and `OutputStream`.
  - c. When finished, close the new client socket connection using the `close()` method of `Socket`.

Our next example, `TCPEchoServer.java`, implements the echo service used by our client program. The server is very simple. It runs forever, repeatedly accepting a connection, receiving and echoing bytes until the connection is closed by the client, and then closing the client socket.

#### ***TCPEchoServer.java***

```

0 import java.net.*; // for Socket, ServerSocket, and InetAddress
1 import java.io.*; // for IOException and Input/OutputStream

```

```

2
3 public class TCPEchoServer {
4
5 private static final int BUFSIZE = 32; // Size of receive buffer
6
7 public static void main(String[] args) throws IOException {
8
9 if(args.length!=1)//Test for correct # of args
10 throw new IllegalArgumentException("Parameter(s):<Port>");
11
12 int servPort=Integer.parseInt(args[0]);
13
14 //Create a server socket to accept client connection requests
15 ServerSocket servSock=newServerSocket(servPort);
16
17 int recvMsgSize;//Size of received message
18 byte[] receiveBuf=new byte[BUFSIZE];//Receive buffer
19
20 while(true){//Run forever, accepting and servicing connections
21     Socket clntSock=servSock.accept();//Get client connection
22
23     SocketAddress clientAddress=clntSock.getRemoteSocketAddress();
24     System.out.println("Handling client at"+clientAddress);
25
26     InputStream in=clntSock.getInputStream();
27     OutputStream out=clntSock.getOutputStream();
28
29     //Receive until client closes connection, indicated by -1 return
30     while((recvMsgSize=in.read(receiveBuf))!=-1){
31         out.write(receiveBuf,0,recvMsgSize);
32     }
33     clntSock.close();//Close the socket. We are done with this client!
34 }
35 /*NOTREACHED*/
36 }
37 }
```

1 Application setup and parameter parsing: lines 0–12  
2 Server socket creation: line 15. *servSock* listens for client connection requests on the port specified in the constructor.  
3 Loop forever, iteratively handling incoming connections: lines 20–34

- Accept an incoming connection: line 21. The sole purpose of a ServerSocket instance is to supply a new, connected Socket instance for each new incoming TCP connection. When the server is ready to handle a client, it calls `accept()`, which blocks until an incoming connection is made to the ServerSocket's port. (If a connection arrives between the time the server socket is constructed and the call to `accept()`, the new connection is queued, and in that case `accept()` returns immediately.) The `accept()` method of ServerSocket returns an instance of Socket that is already connected to the client's remote socket and ready for reading and writing.
- Report connected client: lines 23–24 We can query the newly created Socket instance for the address and port of the connecting client. The `getRemoteSocketAddress()` method of Socket returns an instance of InetSocketAddress that contains the address and port of the client. The `toString()` method of InetSocketAddress prints the information in the form “/address:port”. (The name part is empty because the instance was created from the address information only.)
- Get socket input and output streams: lines 26–27. Bytes written to this socket's OutputStream will be read from the client's socket's InputStream, and bytes written to the client's OutputStream will be read from this socket's InputStream.
- Receive and repeat data until the client closes: lines 30–32. The while loop repeatedly reads bytes (when available) from the input stream and immediately writes the same bytes back to the output stream until the client closes the connection. The `read()` method of InputStream fetches up to the maximum number of bytes the array can hold (in this case, BUFSIZE bytes) into the byte array (*receiveBuf*) and returns the number of bytes read. `read()` blocks until data is available and returns -1 if there is no more data available, indicating that the client closed its socket. In the echo protocol, the client closes the connection when it has received the number of bytes that it sent, so in the server we expect to eventually receive a -1 from `read()`. (Recall that in the client, receiving a -1 from `read()` indicates a protocol error, because it can only happen if the server prematurely closed the connection.)
- As previously mentioned, `read()` does not have to fill the entire byte array to return. In fact, it can return after having read only a single byte. This `write()` method of OutputStream writes *recvMsgSize* bytes from *receiveBuf* to the socket. The second parameter indicates the offset into the byte array of the first byte to send. In this case, 0 indicates to take bytes starting from the front of *data*. If we had used the form of `write()` that takes only the buffer argument, *all* the bytes in the buffer array would have been transmitted, possibly including bytes that were not received from the client.
- Close client socket: line 33. Closing the socket releases system resources associated with the connection, and is required for servers, because there is a system-specific limit on the number of open Socket instances a program can have.

#### *ServerSocket:Creation*

ServerSocket(int localPort)

```

ServerSocket(int localPort, int queueLimit)
ServerSocket(int localPort, int queueLimit, InetAddress localAddr)
ServerSocket()

```

A TCP endpoint must be associated with a specific port in order for clients to direct their connections to it. The first three constructors create a TCP endpoint that is associated with the specified local port and ready to *accept* incoming connections. Valid port numbers are in the range 0–65,535. (If the port specified is zero, an arbitrary unused port will be picked.) Optionally, the size of the connection queue and the local address can also be set. Note that the maximum queue size may not be a hard limit, and cannot be used to control client population. The local address, if specified, must be an address of one of this host's network interfaces. If the address is not specified, the socket will accept connections to any of the host's IP addresses. This may be useful for hosts with multiple interfaces where the server wants to accept connections on only one of its interfaces. The fourth constructor creates a ServerSocket that is not associated with any local port; it must be *bound* to a port (see *bind()* below) before it can be used.

#### *ServerSocket: Operation*

```

void bind(int port)
void bind(int port, int queueLimit)
Socket accept()
void close()

```

The *bind()* methods associate this socket with a local port. A ServerSocket can only be associated with one port. If this instance is already associated with another port, or if the specified port is already in use, an *IOException* is thrown. *accept()* returns a connected *Socket* instance for the next new incoming connection to the server socket. If no established connection is waiting, *accept()* blocks until one is established or a timeout occurs. The *close()* method closes the socket. After invoking this method, incoming client connection requests for this socket are rejected.

#### *ServerSocket: Getting attributes*

```

InetAddress getInetAddress()
SocketAddress getLocalSocketAddress()
int getLocalPort()

```

These return the local address/port of the server socket. Note that, unlike a *Socket*, a *ServerSocket* has no associated I/O Streams. It does, however, have other attributes called options, which can be controlled via various methods.

### 3.2.3.4 Another Example

A communication link created via TCP/IP sockets is a connection-orientated link. This means that the connection between server and client remains open throughout the duration of the dialogue between the two and is only broken (under normal circumstances) when one end of the dialogue formally terminates the exchanges (via an agreed protocol). Since there are two separate types of process involved (client and server), we shall examine them separately, taking the server first. Setting up a server process requires five steps.

#### 1. Create a *ServerSocket* object.

The *ServerSocket* constructor requires a port number (1024–65535, for non-reserved ones) as an argument. For example:

```
ServerSocket servSock = new ServerSocket(1234);
```

In this example, the server will await ('listen for') a connection from a client on port 1234.

#### 2. Put the server into a waiting state.

The server waits indefinitely ('blocks') for a client to connect. It does this by calling method *accept* of class *ServerSocket*, which returns a *Socket* object when a connection is made. For example:

```
Socket link = servSock.accept();
```

#### 3. Set up input and output streams.

Methods *getInputStream* and *getOutputStream* of class *Socket* are used to get references to streams associated with the socket returned in step 2. These streams will be used for communication with the client that has just made connection. For a non-GUI application, we can wrap a *Scanner* object around the *InputStream* object returned by method *getInputStream*, in order to obtain string-orientated input (just as we would do with input from the standard input stream, *System.in*). For example:

```
Scanner input = new Scanner(link.getInputStream());
```

Similarly, we can wrap a *PrintWriter* object around the *OutputStream* object returned by method *getOutputStream*. Supplying the *PrintWriter* constructor with a second argument of true will cause the output

buffer to be flushed for every call of *println* (which is usually desirable). For example:

```
PrintWriter output = new PrintWriter(link.getOutputStream(), true);
```

#### 4. Send and receive data.

Having set up our *Scanner* and *PrintWriter* objects, sending and receiving data is very straightforward. We simply use method *nextLine* for receiving data and method *println* for sending data, just as we might do for console I/O. For example:

```
output.println("Awaiting data...");  
String input = input.nextLine();
```

#### 5. Close the connection (after completion of the dialogue).

This is achieved via method *close* of class *Socket*. For example:

```
link.close();
```

The following example program is used to illustrate the use of these steps.

#### Example

In this simple example, the server will accept messages from the client and will keep count of those messages, echoing back each (numbered) message. The main protocol for this service is that client and server must alternate between sending and receiving (with the client initiating the process with its opening message, of course). The only details that remain to be determined are the means of indicating when the dialogue is to cease and what final data (if any) should be sent by the server. For this simple example, the string "\*\*\*\*CLOSE\*\*\*" will be sent by the client when it wishes to close down the connection. When the server receives this message, it will confirm the number of preceding messages received and then close its connection to this client. The client, of course, must wait for the final message from the server before closing the connection at its own end.

Since an *IOException* may be generated by any of the socket operations, one or more try blocks must be used. Rather than have one large try block (with no variation in the error message produced and, consequently, no indication of precisely what operation caused the problem), it is probably good practice to have the opening of the port and the dialogue with the client in separate try blocks. It is also good practice to place the closing of the socket in a finally clause, so that, whether an exception occurs or not, the socket will be closed (unless, of course, the exception is generated when actually closing the socket, but there is nothing we can do about that). Since the finally clause will need to know about the *Socket* object, we shall have to declare this object within a scope that covers both the try block handling the dialogue and the finally block. Thus, step 2 shown above will be broken up into separate declaration and assignment. In our example program, this will also mean that the *Socket* object will have to be explicitly initialised to null (as it will not be a global variable).

Since a server offering a public service would keep running indefinitely, the call to method *handleClient* in our example has been placed inside an ‘infinite’ loop, thus:

```
do {  
    handleClient();  
} while (true);
```

In the code that follows (and in later examples), port 1234 has been chosen for the service, but it could just as well have been any integer in the range 1024-65535. Note that the lines of code corresponding to each of the above steps have been clearly marked with emboldened comments.

```
//Server that echoes back client's messages.  
//At end of dialogue, sends message indicating number of  
//messages received. Uses TCP.  
import java.io.*;  
import java.net.*;  
import java.util.*;  
public class TCPEchoServer  
{  
    private static ServerSocket servSock;  
    private static final int PORT = 1234;  
    public static void main(String[] args)  
    {
```

```

System.out.println("Opening port...\\n");
try
{
    servSock = new ServerSocket(PORT); //Step 1.
}
catch(IOException ioEx)
{
    System.out.println("Unable to attach to port!"); System.exit(1);
}
do
{
    handleClient();
}while (true);
}

private static void handleClient() {
    Socket link = null; //Step 2.
    try { link = servSock.accept(); //Step 2.
        Scanner input = new Scanner(link.getInputStream()); //Step 3.
        PrintWriter output = new PrintWriter( link.getOutputStream(),true); //Step 3.
        int numMessages = 0; String message = input.nextLine(); //Step 4.
        while (!message.equals("****CLOSE****")) {
            System.out.println("Message received.");
            numMessages++;
            output.println("Message " + numMessages + ": " + message); //Step 4.
            message = input.nextLine();
        }
        output.println(numMessages + " messages received.");//Step 4.
    } catch(IOException ioEx) {
        ioEx.printStackTrace();
    }
    finally
    {
        try
        {
            System.out.println("\n* Closing connection... *");
            link.close(); //Step 5.
        } catch(IOException ioEx) {
            System.out.println("Unable to disconnect!"); System.exit(1);
        }
    } } }

```

Setting up the corresponding client involves four steps.

### *1. Establish a connection to the server.*

We create a *Socket* object, supplying its constructor with the following two arguments:

- the server's IP address (of type *InetAddress*);
- the appropriate port number for the service. (The port number for server and client programs must be the same, of course!)

For simplicity's sake, we shall place client and server on the same host, which will allow us to retrieve the IP address by calling static method *getLocalHost* of class *InetAddress*. For example:

```
Socket link = new Socket(InetAddress.getLocalHost(),1234);
```

### *2. Set up input and output streams.*

These are set up in exactly the same way as the server streams were set up (by calling methods *getInputStream* and *getOutputStream* of the *Socket* object that was created in step 2).

### *3. Send and receive data.*

The *Scanner* object at the client end will receive messages sent by the *PrintWriter* object at the server end, while the *PrintWriter* object at the client end will send messages that are received by the *Scanner* object at the server

end (using methods *nextLine* and *println* respectively).

#### 4. Close the connection.

This is exactly the same as for the server process (using method *close* of class *Socket*).

The code below shows the client program for our example. In addition to an input stream to accept messages from the server, our client program will need to set up an input stream (as another *Scanner* object) to accept user messages from the keyboard. As for the server, the lines of code corresponding to each of the above steps have been clearly marked with emboldened comments.

```
import java.io.*;
import java.net.*;
import java.util.*;
public class TCPEchoClient
{
    private static InetAddress host;
    private static final int PORT = 1234;
    public static void main(String[] args)
    {
        try
        {
            host = InetAddress.getLocalHost();
        } catch(UnknownHostException uhEx) {
            System.out.println("Host ID not found!"); System.exit(1);
        accessServer();
    }
    private static void accessServer() { Socket
link = null; //Step 1.
try { link = new Socket(host,PORT); //Step 1.
Scanner input = new Scanner(link.getInputStream()); //Step 2.
PrintWriter output = new PrintWriter(link.getOutputStream(),true);
//Step 2.
//Set up stream for keyboard entry...
Scanner userEntry = new Scanner(System.in);
String message, response;
do
{
    System.out.print("Enter message: ");
    message = userEntry.nextLine(); output.println(message); //Step 3.
    response = input.nextLine(); //Step 3.
    System.out.println("\nSERVER> "+response);
}while (!message.equals("****CLOSE****"));
}
catch(IOException ioEx) { ioEx.printStackTrace(); }
finally
{
    try
    {
        System.out.println( "\n* Closing connection... *");
        link.close(); //Step 4.
    } catch(IOException ioEx) {
        System.out.println("Unable to disconnect!"); System.exit(1); } } } }
```

For the preceding client-server application to work, TCP/IP must be installed and working. How are you to know whether this is the case for your machine? Well, if there is a working Internet connection on your machine, then TCP/IP is running. In order to start the application, first open two command windows and then start the server running in one window and the client in the other (Make sure that the server is running first).

## 3.2.4 Streams, Readers, and Writers for Input and Output

### 3.2.4.1 Input and Output Streams

The basic I/O paradigm for TCP sockets in Java is the *stream* abstraction. (The NIO facilities, added in Java 1.4, provide an alternative abstraction). A stream is simply an ordered sequence of bytes. Java *input streams* support reading bytes, and *output streams* support writing bytes. In our TCP client and server, each Socket instance holds an InputStream and an OutputStream instance. When we write to the output stream of a Socket, the bytes can (eventually) be read from the input stream of the Socket at the other end of the connection.

OutputStream is the abstract superclass of all output streams in Java. Using an OutputStream, we can write bytes to, flush, and close the output stream.

*OutputStream: Operation*

```
abstract void write(int data)
void write(byte[ ] data)
void write(byte[ ] data, int offset, int length)
void flush()
void close()
```

The write() methods transfer to the output stream a single byte, an entire array of bytes, and the bytes in an array beginning at offset and continuing for length bytes, respectively. The single-byte method writes the low-order eight bits of the integer argument. These operations, if called on a stream associated with a TCP socket, may block if a lot of data has been sent, but the other end of the connection has not called read() on the associated input stream recently. This can have undesirable consequences if some care is not used. The flush() method pushes any buffered data out to the output stream. The close() method terminates the stream, after which further calls to write() will throw an exception.

InputStream is the abstract superclass of all input streams. Using an InputStream, we can read bytes from and close the input stream.

*InputStream: Operation*

```
abstract int read()
int read(byte[ ] data)
int read(byte[ ] data, int offset, int length)
int available()
void close()
```

The first three methods get transfer data from the stream. The first form places a single byte in the low-order eight bits of the returned int. These second form transfers up to *data.length* bytes from the input stream into *data* and returns the number of bytes transferred. The third form does the same, but places data in the array beginning at offset, and transfers only up to length bytes. If no data is available, but the end-of-stream has not been detected, all the read() methods block until at least one byte can be read. All methods return -1 if called when no data is available and end-of-stream has been detected. The available() method returns the number of bytes available for reading at the time it was called. close() shuts down the stream, causing further attempts to read to throw an IOException.

### 3.2.4.2 Readers and Writers

Once we make a connection between two processes over the network, we need a simple, easy way to send and receive data in different formats over the connection. Java provides this through the stream classes in the java.io package. Included in the java.io package are the InputStream and OutputStream classes and their subclasses for byte-based I/O, and the Reader and Writer classes and their subclasses for character-based I/O. The InputStream and OutputStream classes handle data as bytes, with basic methods for reading and writing bytes and byte arrays. Their subclasses can connect to various sources and destinations (files, string buffers), and provide methods for directly sending and receiving basic Java data types, like floating-point values. The Reader and Writer classes transmit data in the form of 16-bit Unicode characters, which provides a platform-independent way to send and receive textual data. Like the InputStream and OutputStream subclasses, the subclasses of Reader and Writer specialize in terms of their source and destination types.

A Socket, once it's created, can be queried for its input/output streams using `getInputStream()` and `getOutputStream()`. These methods return instances of InputStream and OutputStream, respectively. If you need to exchange mostly character-based data between two agents in your distributed system, then you can wrap the InputStream with an InputStreamReader (a subclass of Reader), or the OutputStream with an OutputStreamWriter (a subclass of Writer).

Another way to create an interprocess communication link is to use the `java.lang.Runtime` interface to execute a

process, then obtain the input and output streams from the returned Process object, as shown in Example. You would do this if you had a local subtask that needed to run in a separate process, but with which you still needed to exchange messages.

### Example 3.1. Interprocess I/O Using Runtime-Executed Processes

```
Runtime r = Runtime.getRuntime();
Process p = r.exec("/bin/ls /tmp");
InputStream in = p.getInputStream();
OutputStream out = p.getOutputStream();
```

From the abstract I/O classes, the java.io package offers several specializations which vary the format of the data transmitted over the stream, as well as the type of data source/receiver at the ends of the stream. The InputStream, OutputStream, Reader, and Writer classes provide basic interfaces for data I/O (read() and write() methods that just transfer bytes, byte arrays, characters and character arrays). To define data types and communication protocols on top of these base classes, Java offers the FilterInputStream and FilterOutputStream classes for byte-oriented I/O, and the FilterReader and FilterWriter for character-based I/O. Subclasses of these offer a higher level of control and structure to the data transfers. A BufferedInputStream or BufferedReader uses a memory buffer for efficient reading of data. The overhead associated with data read requests is minimized by performing large data reads into a buffer, and offering data to the caller from the local buffer until it's been exhausted. This feature can be used to minimize the latency associated with slow source devices and communication media. The BufferedOutputStream or BufferedWriter performs the same service on outgoing data. A PushbackInputStream or PushbackReader provides a buffer for pushing back data onto the incoming data stream. This is useful in parsing applications, where the next branch in the parse tree is determined by peeking at the next few bytes or characters in the stream, and then letting the subparser operate on the data. The other interesting subclasses of FilterInputStream and FilterOutputStream are the DataInputStream and DataOutputStream classes. These classes read and write Java data primitives in a portable binary format. There aren't similar subclasses of FilterReader and FilterWriter, since Readers and Writers only transfer character data, and the serialized form of Java data types are represented in bytes.

Besides being useful in their own right for manipulating and formatting input/output data streams, the subclasses of FilterInputStream, FilterOutputStream, FilterReader, and FilterWriter are also well suited for further specialization to define application-specific data stream protocols. Each of the stream classes offers a constructor method, which accepts an InputStream or OutputStream as an argument. Likewise, the FilterReader class has a constructor that accepts a Reader, and FilterWriter has a constructor that accepts a Writer object. In each case, the constructor argument is taken as the source or sink of the stream that is to be filtered, which enables the construction of stream filter "pipelines." So defining a special-purpose data protocol is simply a matter of subclassing from an appropriate I/O class, and wrapping an existing data source or sink with the new filter.

For example, if we wanted to read an XDR-formatted data stream (XDR is the binary format underlying Remote Procedure Call (RPC) data connections), we could write a subclass of FilterInputStream that would offer the same methods to read Java primitive data types as DataInputStream, but would be implemented to parse the XDR format, rather than the portable binary format of the DataInputStream. The following Example shows a skeleton for the input version of this kind of stream; it shows a sample application using the stream. The application first connects to a host and port, where presumably another process is waiting to accept this connection. The remote process uses XDR-formatted data to communicate, so we wrap the input stream from the socket connection with our XDRInputStream and begin reading data.

### Example 3.2. An InputStream Subclass for Reading XDR-Formatted Data

```
package dcj.examples;
import java.io.*;
import java.net.*;
class XDRInputStream extends FilterInputStream {
    public XDRInputStream(InputStream in) {
        super(in);
    }
    // Overridden methods from FilterInputStream, implemented
    // to read XDR-formatted data
    public boolean readBoolean() throws IOException;
    public byte readByte() throws IOException;
    public int readUnsignedByte() throws IOException;
    public float readFloat() throws IOException;
```

```

// Other readXXX() methods omitted in this example...
// We'll assume this stream doesn't support mark/reset operations
    public boolean markSupported() { return false; }
}

Example 3.3. Example XDRInputStream Client
import dcj.examples.XDRInputStream;
import java.io.*;

class XDRInputExample
{
    public static void main(String argv[])
    {
        String host = argv[0];
        // Default port is 5001
        int port = 5001;
        try {
            port = Integer.parseInt(argv[1]);
        } catch (NumberFormatException e) {
            System.out.println("Bad port number given, using default " + port);
        }
        // Try connecting to specified host and port
        Socket serverConn = null;
        try { serverConn = new Socket(host, port); }
        catch (UnknownHostException e)
        {
            System.out.println("Bad host name given.");
            System.exit(1);
        }
        // Wrap an XDR stream around the input stream
        XDRInputStream xin = new XDRInputStream(serverConn.getInputStream());
        // Start reading expected data from XDR-formatted stream
        int numVals = xin.readInt();
        float val1 = xin.readFloat();
        ...
    }
}

```

The classes in the `java.io` package also offer the ability to specialize the sources and destinations of data. The following Table 3.2 summarizes the various stream, writer, and reader classes in `java.io`, and the types of sources and destinations that they can access. The purpose and use of the file, byte–array, and string classes are fairly obvious, and we won't spend any time going into detail about them here, since we'll see them being used in some of the examples later in the book. The stream classes that allow communication between threads deserve some explanation, though.

**Table 3.2. Source and Destination Types Supported by `java.io`**

Source/Destination Type	Input/OutputStream Class	Reader/Writer Class
Remote or local process	InputStream OutputStream (created from Socket or from Process)	InputStream Reader OutputStream Writer
Disk files	FileInputStream FileOutputStream	FileReader FileWriter
In-memory data buffers	ByteArrayInputStream ByteArrayOutputStream	CharArrayReader CharArrayWriter
In-memory string buffers	StringBufferInputStream (input only) (use StringTokenizer)	StringReader StringWriter
Threads within same process	PipedInputStream PipedOutputStream	PipedReader PipedWriter

The `PipedInputStream` and `PipedOutputStream` classes access data from each other. That is, a `PipedInputStream`

reads data from a PipedOutputStream, and a PipedOutputStream writes data to a PipedInputStream. This class design allows the developer to establish data pipes between threads in the same process. Example 3.4 and 3.5 show client and server classes that use piped streams to transfer information, and Example 3.6 shows an application of these classes.

#### Example 3.4. A Piped Client

```
package dcj.examples;
import java.lang.*;
import java.net.*;
import java.io.*;
import java.util.*;

public class PipedClient extends Thread
{
    PipedInputStream pin;
    PipedOutputStream pout;

    public PipedClient(PipedInputStream in, PipedOutputStream out)
    {
        pin = in;
        pout = out;
    }

    public void run()
    {
        // Wrap a data stream around the input and output streams
        DataInputStream din = new DataInputStream(pin);
        DataOutputStream dout = new DataOutputStream(pout);

        // Say hello to the server...
        try
        {
            System.out.println("PipedClient:Writing greeting to server...");
            dout.writeChars("hello from PipedClient\n");
        }
        catch (IOException e)
        {
            System.out.println("PipedClient: Couldn't get response.");
            System.exit(1);
        }

        // See if it says hello back...
        try
        {
            System.out.println("PipedClient: Reading response from server...");
            String response = din.readLine();
            System.out.println("PipedClient: Server said: \\" + response + "\\");
        }
        catch (IOException e)
        {
            System.out.println("PipedClient: Failed to connect to peer.");
        }
    }

    stop();
}
}
```

The example shows two threads, a client and a server, talking to each other over piped streams. The PipedClient class accepts a PipedInputStream and PipedOutputStream as constructor arguments; the PipedServer class does the same. Both are extensions of the Thread class. The client attempts to send a "hello" message to the server over its output stream, then listens for a response on its input stream. The server listens for the "hello" from the client on its input stream, then sends a response back on its output stream. The PipedStreamExample class sets up the stream connections for the threads by creating two pairs of piped streams. It then creates a PipedClient and a PipedServer, sends each the input stream from one pair and the output stream from the other, and tells each of

them to start their threads. The important feature of this example is that the piped streams are connected to each other within the same process, and are not connected to any remote hosts.

### Example 3.5. A Piped Server

```
package dcj.examples;
import java.lang.*;
import java.net.*;
import java.io.*;

public class PipedServer extends Thread
{
    PipedInputStream pin;
    PipedOutputStream pout;

    public PipedServer(PipedInputStream in, PipedOutputStream out)
    {
        pin = in;
        pout = out;
    }
    public void run()
    {
        // Wrap a data stream around the input and output streams
        DataInputStream din = new DataInputStream(pin);
        DataOutputStream dout = new DataOutputStream(pout);

        // Wait for the client to say hello...
        try
        {
            System.out.println("PipedServer: Reading from client...");
            String clientHello = din.readLine();
            System.out.println("PipedServer: Client said:\\" + 
                clientHello+"\"");
        }
        catch (IOException e)
        {
            System.out.println("PipedServer: Couldn't get hello from client.");
            stop();
        }
        // ...and say hello back.
        try
        {
            System.out.println("PipedServer: Writing response to client...");
            dout.writeChars("hello I am the server.\n");
        }
        catch (IOException e)
        {
            System.out.println("PipedServer: Failed to
connect to client.");
        }
        stop();
    }
}
```

### Example 3.6. Piped Stream Application

```
package dcj.examples;

import java.net.*;
import java.io.*;
import java.lang.*;
import dcj.examples.PipedClient;
import dcj.examples.PipedServer;

class PipedStreamExample {
```

```

public static void main(String argv[]) {
    // Make two pairs of connected piped streams
    PipedInputStream pinc = null;
    PipedInputStream pins = null;
    PipedOutputStream poutc = null;
    PipedOutputStream pouts = null;

    try {
        pinc = new PipedInputStream();
        pins = new PipedInputStream();
        poutc = new PipedOutputStream(pins);
        pouts = new PipedOutputStream(pinc);
    }
    catch (IOException e) {
        System.out.println(
            "PipedStreamExample: Failed to build piped streams.");
        System.exit(1);
    }

    // Make the client and server threads, connected by the streams
    PipedClient pc = new PipedClient(pinc, poutc);
    PipedServer ps = new PipedServer(pins, pouts);

    // Start the threads
    System.out.println("Starting server...");
    ps.start();
    System.out.println("Starting client...");
    pc.start();
    // Wait for threads to end

    try {
        ps.join();
        pc.join();
    }
    catch (InterruptedException e) {}
    System.exit(0);
}
}

```

Note that a similar scenario could be set up using the `PipedReader` and `PipedWriter` classes, if you knew the two threads were going to exchange character arrays.

### 3.2.4.3 Another Example

To demonstrate the network support in Java and how it can be exploited for distributed applications, Examples 3.7 to 3.11 show an implementation of this simple client–server system using sockets and input/output streams. The implementation includes the following elements:

- A set of command objects that represent our command protocol between the client and the server
- A subclass of `java.io.DataInputStream` that understands our protocol
- A client that can send commands in the right format to the server, and a server that can accept client connections, read commands using our specialized stream, and send responses back

The client connects to the server over a socket, then sends commands to the server over the socket. The server uses the specialized `DataInputStream` to read the commands from the socket. The input stream automatically creates the right command object based on the type of message from the client (e.g., a "GET" message will be converted into a `GetCmd` object). The server then executes the command and sends the result to the client over the socket.

Example JDC.E1 shows a set of classes that represent the commands a client can send to our server. The `SimpleCmd` class simply holds a single `String` argument and has an abstract `Do()` method that subclasses will implement to do the right thing for the particular command they represent. Our protocol consists of three basic commands: "GET," "HEAD," and "POST," along with a command to close the connection, "DONE." The `GetCmd`, `HeadCmd`, `PostCmd`, and `DoneCmd` classes derived from `SimpleCmd` represent these commands.

### Example 3.7. Commands for the Client–Server System

```
package dcj.examples;
import java.lang.*;
abstract class SimpleCmd
{
    protected String arg;
    public SimpleCmd(String inArg) {
        arg = inArg;
    }
    public abstract String Do();
}
class GetCmd extends SimpleCmd
{
    public GetCmd(String s) { super(s); }
    public String Do() {
        String result = arg + " Gotten\n";
        return result;
    }
}
public class HeadCmd extends SimpleCmd
{
    public HeadCmd(String s) { super(s); }
    public String Do() {
        String result = "Head \"\" " + arg + "\" processed.\n";
        return result;
    }
}
class PostCmd extends SimpleCmd
{
    public PostCmd(String s) { super(s); }
    public String Do() {
        String result = arg + " Posted\n";
        return result;
    }
}
class DoneCmd extends SimpleCmd
{
    public DoneCmd() { super(""); }
    public String Do() {
        String result = "All done.\n";
        return result;
    }
}
```

The classes in Example 3.7 represent the communication protocol for our client–server application, and the SimpleCmdInputStream class acts as the communication link that understands this protocol. The SimpleCmdInputStream is a subclass of java.io.DataInputStream that adds a readCommand() method to its interface. This method parses the data coming in over the stream, determines which command is being sent.

### Example 3.8 A Specialized DataInputStream

```
package dcj.examples;
import java.lang.*;
import java.io.*;
import java.net.*;
public class SimpleCmdInputStream extends DataInputStream
{
    public SimpleCmdInputStream(InputStream in) {
        super(in);
    }
    public String readString() throws IOException {
        StringBuffer strBuf = new StringBuffer();
        boolean hitSpace = false;
        while (!hitSpace) {
            char c = readChar();
            hitSpace = Character.isSpace(c);
```

```

        if (!hitSpace)
            strBuf.append(c);
    }
    String str = new String(strBuf);
    return str;
}
public SimpleCmd readCommand() throws IOException {
    SimpleCmd cmd;
    String commStr = readString();
    if (commStr.compareTo("HEAD") == 0)
        cmd = new HeadCmd(readString());
    else if (commStr.compareTo("GET") == 0)
        cmd = new GetCmd(readString());
    else if (commStr.compareTo("POST") == 0)
        cmd = new PostCmd(readString());
    else if (commStr.compareTo("DONE") == 0)
        cmd = new DoneCmd();
    else
        throw new IOException("Unknown command.");
    return cmd;
}
}

```

Finally, the `SimpleClient` and `SimpleServer` serve as the client and server agents in our distributed system. Our `SimpleClient` is very simple indeed. In its constructor, it opens a socket to a server on a given host and port number. Its `main()` method makes a `SimpleClient` object using command-line arguments that specify the host and port to connect to, then calls the `sendCommands()` method on the client. This method just sends a few commands in the right format to the server over the `OutputStream` from the socket connection.

Notice that the client's socket is closed in its `finalize()` method. This method will only get called after all references to the client are gone, and the system garbage-collector runs to mark the object as finalizable. If it's important that the socket be closed immediately after the client is done with it, you may want to close the socket explicitly at the end of the `sendCommands()` method.

### Example 3.9 A Simple Client

```

package dcj.examples;
import java.lang.*;
import java.net.*;
import java.io.*;
public class SimpleClient
{
    // Our socket connection to the server
    protected Socket serverConn;
    // The input command stream from the server
    protected SimpleCmdInputStream inStream;
    public SimpleClient(String host, int port)
        throws IllegalArgumentException {
        try {
            System.out.println("Trying to connect to " + host + " " + port);
            serverConn = new Socket(host, port);
        }
        catch (UnknownHostException e) {
            throw new IllegalArgumentException("Bad host name given.");
        }
        catch (IOException e) {
            System.out.println("SimpleClient: " + e);
            System.exit(1);
        }
        System.out.println("Made server connection.");
    }
    public static void main(String argv[]) {
        if (argv.length < 2) {
            System.out.println("Usage: java SimpleClient [host] [port]");
            System.exit(1);
        }
    }
}

```

```

String host = argv[0];
int port = 3000;
try {
    port = Integer.parseInt(argv[1]);
}
catch (NumberFormatException e) {}
SimpleClient client = new SimpleClient(host, port);
client.sendCommands();
}
public void sendCommands() {
try {
    OutputStreamWriter wout =
        new OutputStreamWriter(serverConn.getOutputStream());
    BufferedReader rin = new BufferedReader(
        new InputStreamReader(serverConn.getInputStream()));
    // Send a GET command...
    wout.write("GET goodies ");
    // ...and receive the results
    String result = rin.readLine();
    System.out.println("Server says: \"" + result + "\"");
    // Now try a POST command
    wout.write("POST goodies ");
    // ...and receive the results
    result = rin.readLine();
    System.out.println("Server says: \"" + result + "\"");
    // All done, tell the server so
    wout.writeChars("DONE ");
    result = rin.readLine();
    System.out.println("Server says: \"" + result + "\"");
}
catch (IOException e) {
    System.out.println("SimpleClient: " + e);
    System.exit(1);
}
}
public synchronized void finalize() {
System.out.println("Closing down SimpleClient...");
try { serverConn.close(); }
catch (IOException e) {
    System.out.println("SimpleClient: " + e);
    System.exit(1);
}
}
}
}

```

The SimpleServer class has a constructor that binds itself to a given port, and a listen() method that continually checks that port for client connections. Its main() method creates a SimpleServer for a port specified with command-line arguments, then calls the server's listen() method. The listen() method loops continuously, waiting for a client to connect to its port. When a client connects, the server creates a Socketto the client, then calls its serviceClient() method to parse the client's commands and act on them. The serviceClient() takes the InputStream from the client socket, and wraps our SimpleCmdInputStream around it. Then the method loops, calling the readCommand() method on the stream to get the client's commands. If the client sends a DONE command, then the loop stops and the method returns. Until then, each command is read from the stream, and the Do() method is called on each. The string returned from the Do()call is returned to the client over the OutputStream from the client socket.

### Example 3.10 A Simple Server

```

package dcj.examples;
import java.net.*;
import java.io.*;
import java.lang.*;

// A generic server that listens on a port and connects to any clients it
// finds. Made to extend Thread, so that an application can have multiple
// server threads servicing several ports, if necessary.

```

```

public class SimpleServer
{
    protected int portNo = 3000; // Port to listen to for clients
    protected ServerSocket clientConnect;

    public SimpleServer(int port) throws IllegalArgumentException {
        if (port <= 0)
            throw new IllegalArgumentException(
                "Bad port number given to SimpleServer constructor.");

        // Try making a ServerSocket to the given port
        System.out.println("Connecting server socket to port...");
        try { clientConnect = new ServerSocket(port); }
        catch (IOException e) {
            System.out.println("Failed to connect to port " + port);
            System.exit(1);
        }
        // Made the connection, so set the local port number
        this.portNo = port;
    }

    public static void main(String argv[]) {
        int port = 3000;
        if (argv.length > 0) {
            int tmp = port;
            try {
                tmp = Integer.parseInt(argv[0]);
            }
            catch (NumberFormatException e) {}
            port = tmp;
        }
        SimpleServer server = new SimpleServer(port);
        System.out.println("SimpleServer running on port " + port + "...");
        server.listen();
    }

    public void listen() {
        // Listen to port for client connection requests.
        try {
            System.out.println("Waiting for clients...");
            while (true) {
                Socket clientReq = clientConnect.accept();
                System.out.println("Got a client...");
                serviceClient(clientReq);
            }
        }
        catch (IOException e) {
            System.out.println("IO exception while listening for clients.");
            System.exit(1);
        }
    }

    public void serviceClient(Socket clientConn) {
        SimpleCmdInputStream inStream = null;
        DataOutputStream outStream = null;
        try {
            inStream = new SimpleCmdInputStream(clientConn.getInputStream());
            outStream = new DataOutputStream(clientConn.getOutputStream());
        }
        catch (IOException e) {
            System.out.println("SimpleServer: Error getting I/O streams.");
        }
        SimpleCmd cmd = null;
        System.out.println("Attempting to read commands...");
        while (cmd == null || !(cmd instanceof DomeCmd)) {
            try { cmd = inStream.readCommand(); }
            catch (IOException e) {
                System.out.println("SimpleServer: " + e);
            }
        }
    }
}

```

```

        System.exit(1);
    }
    if (cmd != null) {
        String result = cmd.Do();
        try { outStream.writeBytes(result); }
        catch (IOException e) {
            System.out.println("SimpleServer: " + e);
            System.exit(1); } } }
public synchronized void finalize() {
    System.out.println("Shutting down SimpleServer running on port " + portNo); }
}

```

We could easily adapt this simple communication scheme to other applications with different protocols. We would just need to define new subclasses of SimpleCmd, and update our SimpleCmdInputStream to parse them correctly. If we wanted to get exotic, we could expand our communication scheme to implement a "meta-protocol" between agents in the system. The first piece of information passed between two agents when they establish a socket connection would be the protocol they want to use to communicate with each other. Using the class download capabilities mentioned in the previous section, we could actually load a subclass of java.io.InputStream over the newly created socket, create an instance of the class, and attach it to the socket itself. We won't indulge ourselves in this exotic exercise in this chapter, however.

What all of this demonstrates is that Java's network support provides a quick way to develop the communication elements of a basic distributed system. Java's other core features, such as platform-independent bytecodes, facilitate the development of more complex network transactions, such as agents dynamically building a protocol for talking to each other by exchanging class definitions. The core Java API also includes built-in support for sharing Java objects between remote agents, with its RMI package. Objects that implement the java.io.Serializable interface can be converted to byte streams and transmitted over a network connection to a remote Java process, where they can be "reconstituted" into copies of the original objects. Other packages are available for using CORBA to distribute objects within a Java distributed application.

### 3.2.4.4 Security

Java provides two dimensions of security for distributed systems: a secure local runtime environment, and the ability to engage in secure remote transactions.

**Runtime environment.** At the same time that Java facilitates the distribution of system elements across the network, it makes it easy for the recipient of these system elements to verify that they can't compromise the security of the local environment. If Java code is run in the context of an applet, then the Java Virtual Machine places rather severe restrictions on its operation and capabilities. It's allowed virtually no access to the local file system, very restricted network access (e.g., it can only open a network connection back to the server it was loaded from), no access to local code or libraries outside of the Java environment, and restricted thread manipulation capabilities, among other things. In addition, any class definitions loaded over the network, whether from a Java applet or a Java application, are subjected to a stringent bytecode verification process, in which the syntax and operations of the bytecodes are checked for incorrect or potentially malicious behavior.

**Secure remote transactions.** This capability of the environment makes it easy to add user authentication and data encryption to establish secure network links, assuming that the basic encryption and authentication algorithms already exist. Suppose, for example, that we wanted to use public key encryption to establish secure, authenticated connections to named agents on remote machines. We can extend the BufferedInputStream and BufferedOutputStream classes in java.ioto authenticate and decrypt incoming data, and to sign and encrypt outgoing data. The following example displays the encrypted input stream.

#### Example 3.11. Encrypted Input Stream

```

import java.io.*;
public abstract class EncryptedInputStream extends BufferedInputStream
{
    public EncryptedInputStream(InputStream in);
        // Assumes the key ID and signature will be embedded
        // in the incoming data
    public EncryptedInputStream(InputStream in, String id);
        // Will only allow communication once identified
        // entity is authenticated with a public key
}

```

```

// Protected methods
public int decrypt(int) throws SecurityException;
public int decrypt(byte[] b) throws SecurityException;
public int decrypt(byte[] b, int off, int len)
    throws SecurityException;
// Public methods
public int read() throws IOException, SecurityException
{
    return decrypt(super.read());
}
public int read(byte[] b) throws IOException, SecurityException
{
    super.read(b);
    return decrypt(b);
}
public int read(byte[] b, int off, int len)
throws IOException, SecurityException
{
    super.read(b, off, len);
    return decrypt(b, off, len);
}
}
}

```

Of course, the example is greatly simplified by the fact that we haven't actually implemented the `EncryptedInputStream.decrypt()` methods, which are at the heart of the matter, since they actually detect key IDs and signatures, look up public keys on some key list in memory or on disk, and decrypt the incoming data stream once the agent at the other end has been authenticated. We've also avoided the issues of data expansion or compression caused by encryption. When an `EncryptedInputStream` is asked to read  $n$  bytes of data, the intention is typically to read  $n$  decrypted bytes. Any change in data size would have to be made opaque by the `decrypt()` methods.

Once we establish an encrypted communications stream with the remote agent, we can layer any kind of data protocol we like on top of it. For example, our simple GET/HEAD/POST messaging scheme from an earlier example could be carried out securely by simply putting an encrypted input/output stream pair underneath:

```

public SecureClientServer
{
    public SecureClientServer(String host, int port)
        throws SecurityException
    {
        Socket comm = new Socket(host, port);
        InputStream rawIn = comm.getInputStream();
        EncryptedInputStream secureIn = new EncryptedInputStream(rawIn);
        SimpleMessageInputStream msgIn=new SimpleMessageInputStream(secureIn);
        // Start reading and processing commands from the
        // (now encrypted) input stream
        while (true)
        {
            try {
                SimpleCmd cmd = msgIn.readCommand();
                cmd.Do();
            }
            catch (IOException e) {}
        // Remainder of class implementation omitted
        ...
    }
}

```

Of course, this assumes that the agent at the other end of the socket has been suitably augmented to encrypt and decrypt streams.

These examples have simply alluded to how the basic network capabilities of Java could be extended to support secure network communications. The `java.security` package provides a framework for implementing the authentication and encryption algorithms needed to complete our secure input stream example. The authentication process could be implemented using the `KeyPair` and `Signature` classes, for example.

### 3.2.5 UDP Sockets

Most Internet applications use TCP as their transport mechanism. Unfortunately, the checks built into TCP to make it such a robust protocol do not come without a cost. The overhead of providing facilities such as confirmation of receipt and re-transmission of lost or corrupted packets means that TCP is a relatively slow transport mechanism. For many applications (e.g., file transfer), this does not really matter greatly. For these applications, it is much more important that the data arrives intact and in the correct sequence, both of which are guaranteed by TCP. For some applications, however, these factors are not the most important criteria and the relatively slow throughput speed provided by TCP is simply not feasible. Such applications include the playing of audio and video while the associated files are being downloaded, via what is called *streaming*. One of the most popular streaming technologies is called *RealAudio*. *RealAudio* does not use TCP, because of its large overhead. This and other such applications use User Datagram Protocol (UDP). UDP is an unreliable protocol, since:

- it doesn't guarantee that each packet will arrive;
- it doesn't guarantee that packets will be in the right order.

UDP doesn't re-send a packet if it is missing or there is some other error, and it doesn't assemble packets into the correct order. However, it is significantly *faster* than TCP. For applications such as the streaming of audio or video, losing a few bits of data is much better than waiting for re-transmission of the missing data. The major objective in these two applications is to keep playing the sound/video without interruption. In addition, it is possible to build error-checking code into the UDP data streams to compensate for the missing data.

UDP provides an end-to-end service different from that of TCP. In fact, UDP performs only two functions: 1) it adds another layer of addressing (ports) to that of IP, and 2) it detects some forms of data corruption that may occur in transit and discards any corrupted messages. Because of this simplicity, UDP sockets have some different characteristics from the TCP sockets we saw earlier. For example, UDP sockets do not have to be connected before being used. Where TCP is analogous to telephone communication, UDP is analogous to communicating by mail: you do not have to "connect" before you send a package or letter, but you do have to specify the destination address for each one. Similarly, each message—called a *datagram*—carries its own address information and is independent of all others. In receiving, a UDP socket is like a mailbox into which letters or packages from many different sources can be placed. As soon as it is created, a UDP socket can be used to send/receive messages to/from any address and to/from many different addresses in succession.

Another difference between UDP sockets and TCP sockets is the way that they deal with message boundaries: *UDP sockets preserve them*. This makes receiving an application message simpler, in some ways, than it is with TCP sockets. A final difference is that the end-to-end transport service UDP provides is best-effort: there is no guarantee that a message sent via a UDP socket will arrive at its destination, and messages can be delivered in a different order than they were sent (just like letters sent through the mail). A program using UDP sockets must therefore be prepared to deal with loss and reordering.

Given this additional burden, why would an application use UDP instead of TCP? One reason is efficiency: if the application exchanges only a small amount of data—say, a single request message from client to server and a single response message in the other direction—TCP's connection establishment phase at least doubles the number of messages (and the number of round-trip delays) required for the communication. Another reason is flexibility: when something other than a reliable byte-stream service is required, UDP provides a minimal-overhead platform on which to implement whatever is needed.

Unlike TCP/IP sockets, datagram sockets are connectionless. That is to say, the connection between client and server is not maintained throughout the duration of the dialogue. Instead, each datagram packet is sent as an isolated transmission whenever necessary. Since the connection is not maintained between transmissions, the server does not create an individual *Socket* object for each client, as it did in our TCP/IP example. A further difference from TCP/IP sockets is that, instead of a *ServerSocket* object, the server creates a *DatagramSocket* object, as does each client when it wants to send datagram(s) to the server. The final and most significant difference is that *DatagramPacket* objects are created and sent at both ends, rather than simple strings.

Java programmers use UDP sockets via the classes *DatagramPacket* and *DatagramSocket*. Both clients and servers use *DatagramSockets* to send and receive *DatagramPackets*.

#### 3.2.5.1 DatagramPacket

Instead of sending and receiving streams of bytes as with TCP, UDP endpoints exchange self-contained messages, called *datagrams*, which are represented in Java as instances of *DatagramPacket*. To send, a Java program

constructs a DatagramPacket instance containing the data to be sent and passes it as an argument to the send() method of a DatagramSocket. To receive, a Java program constructs a DatagramPacket instance with preallocated space (a byte[]), into which the contents of a received message can be copied (if/when one arrives), and then passes the instance to the receive() method of a DatagramSocket.

In addition to the data, each instance of DatagramPacket also contains address and port information, the semantics of which depend on whether the datagram is being sent or received. When a DatagramPacket is sent, the address and port identify the destination; for a received DatagramPacket, they identify the source of the received message. Thus, a server can receive into a DatagramPacket instance, modify its buffer contents, then send the same instance, and the modified message will go back to its origin. Internally, a DatagramPacket also has *length* and *offset* fields, which describe the location and number of bytes of message data inside the associated buffer.

#### DatagramPacket: Creation

```
DatagramPacket(byte[ ] data, int length)
DatagramPacket(byte[ ] data, int offset, int length)
DatagramPacket(byte[ ] data, int length, InetAddress remoteAddr, int remotePort)
DatagramPacket(byte[ ] data, int offset, int length, InetAddress remoteAddr, int remotePort)
DatagramPacket(byte[ ] data, int length, SocketAddress sockAddr)
DatagramPacket(byte[ ] data, int offset, int length, SocketAddress sockAddr)
```

These constructors create a datagram whose data portion is contained in the given byte array. The first two forms are typically used to construct DatagramPackets for receiving because the destination address is not specified (although it could be specified later with setAddress() and setPort(), or setSocketAddress()). The last four forms are typically used to construct DatagramPackets for sending. Where offset is specified, the data portion of the datagram will be transferred to/from the byte array beginning at the specified position in the array. The length parameter specifies the number of bytes that will be transferred from the bytearray when sending, or the maximum number to be transferred when receiving; it may be smaller, but not larger than data.length. The destination address and port may be specified separately, or together in a SocketAddress.

#### DatagramPacket: Addressing

```
InetAddress getAddress()
void setAddress(InetAddress address)
int getPort()
void setPort(int port)
SocketAddress getSocketAddress()
void setSocketAddress(SocketAddress sockAddr)
```

In addition to constructors, these methods supply an alternative way to access and modify the address of a DatagramPacket. Note that in addition, the receive() method of DatagramSocket sets the address and port to the datagram sender's address and port.

#### DatagramPacket: Handling data

```
int getLength()
void setLength(int length)
int getOffset()
byte[ ] getData()
void setData(byte[ ] data)
void setData(byte[ ] buffer, int offset, int length)
```

The first two methods return/set the internal length of the data portion of the datagram. The internal datagram length can be set explicitly either by the constructor or by the setLength() method. Attempting to make it larger than the length of the associated buffer results in an IllegalArgumentException. The receive() method of DatagramSocket uses the internal length in two ways: on input, it specifies the maximum number of bytes of a received message that will be copied into the buffer and on return, it indicates the number of bytes actually placed in the buffer. getOffset() returns the location in the buffer of the first byte of data to be sent/received. There is no setOffset() method; however, it can be set with setData(). The getData() method returns the byte array associated with the datagram. The returned object is a reference to the byte array that was most recently associated with this DatagramPacket, either by the constructor or by setData(). The length of the returned buffer may be greater than the internal datagram length, so the internal length and offset values should be used to determine the actual received data. The setData() methods make the given byte array the data portion of the datagram. The first form makes the entire byte array the buffer; the second form makes bytes offset through offset+length-1 the buffer. The second form always updates the internal offset and length.

### 3.2.5.2 Example of UDP Client

A UDP client begins by sending a datagram to a server that is passively waiting to be contacted. The typical UDP client goes through three steps:

- 1 Construct an instance of DatagramSocket, optionally specifying the local address and port.
- 2 Communicate by sending and receiving instances of DatagramPacket using the send() and receive() methods of DatagramSocket.

- 3 When finished, deallocate the socket using the close() method of DatagramSocket.

Unlike a Socket, a DatagramSocket is not constructed with a specific destination address. This illustrates one of the major differences between TCP and UDP. A TCP socket is required to establish a connection with another TCP socket on a specific host and port before any data can be exchanged, and, thereafter, it *only* communicates with that socket until it is closed. A UDP socket, on the other hand, is not required to establish a connection before communication, and each datagram can be sent to or received from a different destination. (The connect() method of DatagramSocket does allow the specification of the remote address and port, but its use is optional.)

Our UDP echo client, UDPEchoClientTimeout.java, sends a datagram containing the string to be echoed and prints whatever it receives back from the server. A UDP echo server simply sends each datagram that it receives back to the client. Of course, a UDP client only communicates with a UDP server. Many systems include a UDP echo server for debugging and testing purposes.

One consequence of using UDP is that datagrams can be lost. In the case of our echo protocol, either the echo request from the client or the echo reply from the server may be lost in the network. Recall that our TCP echo client sends an echo string and then blocks on read() waiting for a reply. If we try the same strategy with our UDP echo client and the echo request datagram is lost, our client will block forever on receive(). To avoid this problem, our client uses the setSoTimeout() method of DatagramSocket to specify a maximum amount of time to block on receive(), so it can try again by resending the echo request datagram. Our echo client performs the following steps:

- 1 Send the echo string to the server.
- 2 Block on receive() for up to three seconds, starting over (up to five times) if the reply is not received before the timeout.
- 3 Terminate the client.

#### ***UDPEchoClientTimeout.java***

```

0 import java.net.DatagramSocket;
1 import java.net.DatagramPacket;
2 import java.net.InetAddress;
3 import java.io.IOException;
4 import java.io.InterruptedIOException;
5
6 public class UDPEchoClientTimeout {
7
8     private static final int TIMEOUT = 3000; // Resend timeout (milliseconds)
9     private static final int MAXTRIES = 5; // Maximum retransmissions
10
11    public static void main(String[] args) throws IOException {
12
13        if ((args.length < 2) || (args.length > 3)) { // Test for correct # of args
14            throw new IllegalArgumentException("Parameter(s): <Server> <Word> [<Port>]");
15        }
16        InetAddress serverAddress = InetAddress.getByName(args[0]); // Server address
17        // Convert the argument String to bytes using the default encoding
18        byte[] bytesToSend = args[1].getBytes();
19
20        int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;
21
22        DatagramSocket socket = new DatagramSocket();
23
24        socket.setSoTimeout(TIMEOUT); // Maximum receive blocking time (milliseconds)
25
26        DatagramPacket sendPacket = new DatagramPacket(bytesToSend, // Sending packet
27                                                       bytesToSend.length, serverAddress, servPort);
28
29        DatagramPacket receivePacket = // Receiving packet
30                           new DatagramPacket(new byte[bytesToSend.length], bytesToSend.length);
31
32        int tries = 0; // Packets may be lost, so we have to keep trying
33        boolean receivedResponse = false;
34        do {
35            socket.send(sendPacket); // Send the echo string
36            try {

```

```

37     socket.receive(receivePacket); // Attempt echo reply reception
38
39     if (!receivePacket.getAddress().equals(serverAddress)) { // Check source
40         throw new IOException("Received packet from an unknown source");
41     }
42     receivedResponse = true;
43 } catch (InterruptedException e) { // We did not get anything
44     tries += 1;
45     System.out.println("Timed out, " + (MAXTRIES-tries) + " more tries...");
46 }
47 } while ((!receivedResponse) && (tries < MAXTRIES));
48
49 if (receivedResponse) {
50     System.out.println("Received: " + new String(receivePacket.getData()));
51 } else {
52     System.out.println("No response -- giving up.");
53 }
54 socket.close();
55 }
56 }
```

1 Application setup and parameter processing: lines 0–20.

2 UDP socket creation: line 22. This instance of DatagramSocket can send datagrams to any UDP socket. We do not specify a local address or port so some local address and available port will be selected. We could explicitly set them with the setLocalAddress() and setLocalPort() methods or in the constructor, if desired.

3 Set the socket timeout: line 24. The timeout for a datagram socket controls the maximum amount of time (milliseconds) a call to receive() will block. Here we set the timeout to three seconds. Note that timeouts are not precise: the call may block for more than the specified time (but not less).

4 Create datagram to send: lines 26–27. To create a datagram for sending, we need to specify three things: data, destination address, and destination port. For the destination address, we may identify the echo server either by name or IP address. If we specify a name, it is converted to the actual IP address in the constructor.

5 Create datagram to receive: lines 29–30. To create a datagram for receiving, we only need to specify a byte array to hold the datagram data. The address and port of the datagram source will be filled in by receive().

6 Send the datagram: lines 32–47. Since datagrams may be lost, we must be prepared to retransmit the datagram. We loop sending and attempting a receive of the echo reply up to five times.

- Send the datagram: line 35. send() transmits the datagram to the address and port specified in the datagram.
- Handle datagram reception: lines 36–46. receive() blocks until it either receives a datagram or the timer expires. Timer expiration is indicated by an InterruptedException. If the timer expires, we increment the send attempt count (*tries*) and start over. After the maximum number of tries, the while loop exits without receiving a datagram. If receive() succeeds, we set the loop flag *receivedResponse* to true, causing the loop to exit. Since packets may come from anywhere, we check the source address of the received datagram to verify that it matches the address of the specified echo server.

7 Print reception results: lines 49–53. If we received a datagram, *receivedResponse* is true, and we can print the datagram data.

8 Close the socket: line 54

Before looking at the code for the server, let's take a look at the main methods of the DatagramSocket class.

#### DatagramSocket: Creation

```

DatagramSocket()
DatagramSocket(int localPort)
DatagramSocket(int localPort, InetAddress localAddr)
```

These constructors create a UDP socket. Either or both of the local port and address may be specified. If the local port is not specified, or is specified as 0, the socket is bound to any available local port. If the local address is not specified, the packet can receive datagrams addressed to any of the local addresses.

#### DatagramSocket: Connection and Closing

```

void connect(InetAddress remoteAddr, int remotePort)
void connect(SocketAddress remoteSockAddr)
void disconnect()
void close()
```

The connect() methods set the remote address and port of the socket. Once connected, the socket can only communicate with the specified

address and port; attempting to send a datagram with a different address and port will throw an exception. The socket will only receive datagrams that originated from the specified port and address; datagrams arriving from any other port or address are ignored. A socket connected to a multicast or broadcast address can only *send* datagrams because a datagram source address is always a unicast address. Note that connecting is strictly a local operation because (unlike TCP) there is no end-to-end packet exchange involved. `disconnect()` unsets the remote address and port, if any. The `close()` method indicates that the socket is no longer in use; further attempts to send or receive throw an exception.

#### *DatagramSocket:Addressing*

```
InetAddress getInetAddress()
int getPort()
SocketAddress getRemoteSocketAddress()
InetAddress getLocalAddress()
int getLocalPort()
SocketAddress getLocalSocketAddress()
```

The first method returns an `InetAddress` instance representing the address of the remote socket to which this socket is connected, or null if it is not connected. Similarly, `getPort()` returns the port number to which the socket is connected, or -1 if it is not connected. The third method returns both address and port conveniently encapsulated in an instance of `SocketAddress`, or null if unconnected. The last three methods provide the same service for the *local* address and port. If the socket has not been bound to a local address, `getLocalAddress()` returns the wildcard (“any local address”) address. `getLocalPort()` always returns a local port number; if the socket was not been bound before the call, the call causes the socket to be bound to any available local port. The `getLocalSocketAddress()` returns null if the socket is not bound.

#### *DatagramSocket: Sending and receiving*

```
void send(DatagramPacket packet)
void receive(DatagramPacket packet)
```

The `send()` method sends the `DatagramPacket`. If connected, the packet is sent to the address to which the socket is connected, unless the `DatagramPacket` specifies a different destination, in which case an exception is thrown. Otherwise, the packet is sent to the destination indicated by the `DatagramPacket`. This method does not block. The `receive()` method blocks until a datagram is received, and then copies its data into the given `DatagramPacket`. If the socket is connected, the method blocks until a datagram is received from the remote socket to which it is connected.

#### *DatagramSocket:Options*

```
int getSoTimeout()
void setSoTimeout(int timeoutMillis)
```

These methods return and set, respectively, the maximum amount of time that a `receive()` call will block for this socket. If the timer expires before data is available, an `InterruptedException` is thrown. The timeout value is given in milliseconds.

Like `Socket` and `ServerSocket`, the `DatagramSocket` class has many other options.

### **3.2.5.3 Example of UDP Server**

Like a TCP server, a UDP server’s job is to set up a communication endpoint and passively wait for clients to initiate communication; however, since UDP is connectionless, UDP communication is initiated by a datagram from the client, without going through a connection setup as in TCP. The typical UDP server goes through three steps:

- 1 Construct an instance of `DatagramSocket`, specifying the local port and, optionally, the local address. The server is now ready to receive datagrams from any client.
- 2 Receive an instance of `DatagramPacket` using the `receive()` method of `DatagramSocket`. When `receive()` returns, the datagram contains the client’s address so we know where to send the reply.
- 3 Communicate by sending and receiving `DatagramPackets` using the `send()` and `receive()` methods of `DatagramSocket`.

Our next program example, `UDPEchoServer.java`, implements the UDP version of the echo server. The server is very simple: it loops forever, receiving datagrams and then sending the same datagrams back to the client. Actually, our server only receives and sends back the first 255 (*ECHOMAX*) characters of the datagram; any excess is silently discarded by the socket implementation.

#### ***UDPEchoServer.java***

```
0 import java.io.IOException;
1 import java.net.DatagramPacket;
2 import java.net.DatagramSocket;
3
4 public class UDPEchoServer {
```

```

5
6 private static final int ECHOMAX = 255; // Maximum size of echo datagram
7
8 public static void main(String[] args) throws IOException {
9
10    if (args.length != 1) { // Test for correct argument list
11        throw new IllegalArgumentException("Parameter(s): <Port>");
12    }
13
14    int servPort = Integer.parseInt(args[0]);
15
16    DatagramSocket socket = new DatagramSocket(servPort);
17    DatagramPacket packet = new DatagramPacket(new byte[ECHOMAX], ECHOMAX);
18
19    while (true) { // Run forever, receiving and echoing datagrams
20        socket.receive(packet); // Receive packet from client
21        System.out.println("Handling client at " +
22            packet.getAddress().getHostAddress() + " on port " + packet.getPort());
23        socket.send(packet); // Send the same packet back to client
24        packet.setLength(ECHOMAX); // Reset length to avoid shrinking buffer
25    }
26    /* NOT REACHED */
27 }
28 }
```

1 Application setup and parameter parsing: lines 0–14. UDPEchoServer takes a single parameter, the local port of the echo server socket.

2 Create and set up datagram socket: line 16. Unlike our UDP client, a UDP server must explicitly set its local port to a number known by the client; otherwise, the client will not know the destination port for its echo request datagram. When the server receives the echo datagram from the client, it can find out the client's address and port from the datagram.

3 Create datagram: line 17 UDP messages are contained in datagrams. We construct an instance of DatagramPacket with a buffer of *ECHOMAX* 255 bytes. This datagram will be used both to receive the echo request and to send the echo reply.

4 Iteratively handle incoming echo requests: lines 19–25. The UDP server uses a single socket for all communication, unlike the TCP server, which creates a new socket with every successful accept().

- Receive echo request datagram, print source: lines 20–22. The receive() method of DatagramSocket blocks until a datagram is received from a client (unless a timeout is set). There is no connection, so each datagram may come from a different sender. The datagram itself contains the sender's (client's) source address and port.
- Send echo reply: line 23. *packet* already contains the echo string and echo reply destination address and port, so the send() method of DatagramSocket can simply transmit the datagram previously received. Note that when we receive the datagram, we interpret the datagram address and port as the *source* address and port, and when we send a datagram, we interpret the datagram's address and port as the *destination* address and port.
- Reset buffer size: line 24. The internal length of *packet* was set to the length of the message just processed, which may have been smaller than the original buffer size. If we do not reset the internal length before receiving again, the next message will be truncated if it is longer than the one just received.

### 3.2.5.4 Another Example

Following the style of coverage for TCP client/server applications, the detailed steps required for client and server will be described separately, with the server process being covered first. This process involves the following nine steps, though only the first eight steps will be executed under normal circumstances.

#### 1. Create a DatagramSocket object.

Just as for the creation of a *ServerSocket* object, this means supplying the object's constructor with the port number. For example:

```
DatagramSocket datagramSocket = new DatagramSocket(1234);
```

#### 2. Create a buffer for incoming datagrams.

This is achieved by creating an array of bytes. For example:

```
byte[] buffer = new byte[256];
```

#### 3. Create a DatagramPacket object for the incoming datagrams.

The constructor for this object requires two arguments:

- the previously-created byte array;
- the size of this array.

For example:

```
DatagramPacket inPacket = new DatagramPacket(buffer, buffer.length);
```

#### 4. Accept an incoming datagram.

This is effected via the *receive* method of our *DatagramSocket* object, using our *DatagramPacket* object as the receptacle. For example:

```
datagramSocket.receive(inPacket);
```

#### 5. Accept the sender's address and port from the packet.

Methods *getAddress* and *getPort* of our *DatagramPacket* object are used for this. For example:

```
InetAddress clientAddress = inPacket.getAddress();
int clientPort = inPacket.getPort();
```

#### 6. Retrieve the data from the buffer.

For convenience of handling, the data will be retrieved as a string, using an overloaded form of the *String* constructor that takes three arguments:

- a byte array;
- the start position within the array (= 0 here);
- the number of bytes (= full size of buffer here).

For example:

```
String message = new String(inPacket.getData(), 0, inPacket.getLength());
```

#### 7. Create the response datagram.

Create a *DatagramPacket* object, using an overloaded form of the constructor that takes four arguments:

- the byte array containing the response message;
- the size of the response;
- the client's address;
- the client's port number.

The first of these arguments is returned by the *getBytes* method of the *String* class (acting on the desired *String* response). For example:

```
DatagramPacket outPacket = new DatagramPacket(response.getBytes(),
                                              response.length(), clientAddress, clientPort);
```

(Here, *response* is a *String* variable holding the return message.)

#### 8. Send the response datagram.

This is achieved by calling method *send* of our *DatagramSocket* object, supplying our outgoing *DatagramPacket* object as an argument. For example:

```
datagramSocket.send(outPacket);
```

Steps 4-8 may be executed indefinitely (within a loop).

Under normal circumstances, the server would probably not be closed down at all. However, if an exception occurs, then the associated *DatagramSocket* should be closed, as shown in step 9 below.

## 9. Close the DatagramSocket.

This is effected simply by calling method *close* of our *DatagramSocket* object. For example:

```
datagramSocket.close();
```

To illustrate the above procedure and to allow easy comparison with the equivalent TCP/IP code, the example from previous section will be employed again. As before, the lines of code corresponding to each of the above steps are indicated via emboldened comments. Note that the *numMessages* part of the message that is returned by the server is somewhat artificial, since, in a real-world application, many clients could be making connection and the overall message numbers would not mean a great deal to individual clients. However, the cumulative message-numbering will serve to emphasise that there are no separate sockets for individual clients.

There are two other differences from the equivalent TCP/IP code that are worth noting, both concerning the possible exceptions that may be generated:

- the *IOException* in *main* is replaced with a *SocketException*;
- there is no checked exception generated by the *close* method in the finally clause, so there is no try block.

```
//Server that echoes back client's messages. At end of dialogue, sends message
//indicating number of messages received. Uses datagrams.
```

```
import java.io.*;
import java.net.*;
public class UDPEchoServer
{
    private static final int PORT = 1234;
    private static DatagramSocket datagramSocket;
    private static DatagramPacket inPacket, outPacket;
    private static byte[] buffer;
    public static void main(String[] args)
    {
        System.out.println("Opening port...\\n");
        try
        {
            datagramSocket = new DatagramSocket(PORT);
            //Step 1.
        }
        catch(SocketException sockEx)
        {
            System.out.println( "Unable to attach to port!" );
            System.exit(1);
        }
        handleClient();
    }
    private static void handleClient()
    {
        try {
            String messageIn,messageOut;
            int numMessages = 0;
            do {
                buffer = new byte[256]; //Step 2.
                inPacket = new DatagramPacket( buffer, buffer.length ); //Step 3.
                datagramSocket.receive(inPacket); //Step 4.
                InetAddress clientAddress = inPacket.getAddress(); //Step 5.
                int clientPort = inPacket.getPort(); //Step 5.
                messageIn = new String(inPacket.getData(),0,inPacket.getLength()); //Step 6.
                System.out.println("Message received.");
                numMessages++;
                messageOut = "Message " + numMessages + ":" + messageIn;
                outPacket = new DatagramPacket(messageOut.getBytes(), messageOut.length(),
                    clientAddress, clientPort);
                //Step 7.
                datagramSocket.send(outPacket);
                //Step 8.
            }while (true);
        }
    }
}
```

```

        catch (IOException ioEx)
        {
            ioEx.printStackTrace();
        }
        finally //If exception thrown, close connection.
        {
            System.out.println("\n* Closing connection... *");
            datagramSocket.close(); //Step 9.
        } } }
    
```

Setting up the corresponding client requires the eight steps listed below.

#### *1. Create a DatagramSocket object.*

This is similar to the creation of a DatagramSocket object in the server program, but with the important difference that the constructor here requires no argument, since a default port (at the client end) will be used. For example:

```
DatagramSocket datagramSocket = new DatagramSocket();
```

#### *2. Create the outgoing datagram.*

This step is exactly as for step 7 of the server program. For example:

```
DatagramPacket outPacket = new DatagramPacket(message.getBytes(), message.length(),
host, PORT);
```

#### *8. Send the datagram message.*

Just as for the server, this is achieved by calling method *send* of the *DatagramSocket* object, supplying our outgoing *DatagramPacket* object as an argument. For example:

```
datagramSocket.send(outPacket);
```

Steps 4-6 below are exactly the same as steps 2-4 of the server procedure.

#### *4. Create a buffer for incoming datagrams.*

For example:

```
byte[] buffer = new byte[256];
```

#### *5. Create a DatagramPacket object for the incoming datagrams.*

For example:

```
DatagramPacket inPacket = new DatagramPacket(buffer, buffer.length);
```

#### *6. Accept an incoming datagram.*

For example:

```
datagramSocket.receive(inPacket);
```

#### *7. Retrieve the data from the buffer.*

This is the same as step 6 in the server program. For example:

```
String response = new String(inPacket.getData(), 0, inPacket.getLength());
```

Steps 2-7 may then be repeated as many times as required.

#### *8. Close the DatagramSocket.*

This is the same as step 9 in the server program. For example:

```
datagramSocket.close();
```

As was the case in the server code, there is no checked exception generated by the above *close* method in the finally clause of the client program, so there will be no try block. In addition, since there is no inter-message connection maintained between client and server, there is no protocol required for closing down the dialogue. This means that we do not wish to send the final '**\*\*\*CLOSE\*\*\***' string (though we shall continue to accept this from the user, since we need to know when to stop sending messages at the client end). The line of code (singular, this time) corresponding to each of the above steps will be indicated via an emboldened comment.

```

import java.io.*;
import java.net.*;
import java.util.*;
public class UDPEchoClient
{
    private static InetAddress host;
    private static final int PORT = 1234;
    private static DatagramSocket datagramSocket;
    private static DatagramPacket inPacket, outPacket;
    private static byte[] buffer;
    public static void main(String[] args)
    {
        try
        {
            host = InetAddress.getLocalHost();
        } catch(UnknownHostException uhEx) {
            System.out.println("Host ID not found!");
            System.exit(1);
        }
        accessServer();
    }
    private static void accessServer()
    {
        try
        {
            //Step 1...
            datagramSocket = new DatagramSocket();
            //Set up stream for keyboard entry...
            Scanner userEntry = new Scanner(System.in);
            String message="", response="";
            do
            {
                System.out.print("Enter message: ");
                message = userEntry.nextLine();
                if (!message.equals("/**CLOSE**")) {
                    outPacket = new DatagramPacket( message.getBytes(), message.length(),
                        host,PORT); //Step 2.
                    //Step 3...
                    datagramSocket.send(outPacket);
                    buffer = new byte[256];
                    //Step 4.
                    inPacket = new DatagramPacket(buffer, buffer.length());
                    //Step 5./Step 6...
                    datagramSocket.receive(inPacket);
                    response = new String(inPacket.getData(), 0, inPacket.getLength());
                    //Step 7.
                    System.out.println( "\nSERVER> "+response);
                }
            }while (!message.equals("/**CLOSE**"));
        } catch(IOException ioEx) { ioEx.printStackTrace(); }
        finally
        {
            System.out.println( "\n* Closing connection... *");
            datagramSocket.close();
            //Step 8.
        } } }

```

For the preceding application to work, UDP must be installed and working on the host machine. As for TCP/IP,

if there is a working Internet connection on the machine, then UDP is running. Once again, in order to start the application, first open two command windows and then start the server running in one window and the client in the other. (Start the server *before* the client!)

### 3.2.5.5 Sending and Receiving with UDP Sockets

In this section we consider some of the differences between communicating with UDP sockets compared to TCP. A subtle but important difference is that UDP preserves message boundaries. Each call to receive() on a DatagramSocket returns data from at most one call to send(). Moreover, different calls to receive() will never return data from the same call to send().

When a call to write() on a TCP socket's output stream returns, all the caller knows is that the data has been copied into a buffer for transmission; the data may or may not have actually been transmitted yet. UDP, however, does not provide recovery from network errors and, therefore, does not buffer data for possible retransmission. This means that by the time a call to send() returns, the message has been passed to the underlying channel for transmission and is (or soon will be) on its way out the door.

Between the time a message arrives from the network and the time its data is returned via read() or receive(), the data is stored in a *first-in, first-out (FIFO)* queue of received data. With a connected TCP socket, all received-but-not-yet-delivered bytes are treated as one continuous sequence of bytes. For a UDP socket, however, the received data may have come from different senders. A UDP socket's received data is kept in a queue of messages, each with associated information identifying its source. A call to receive() will never return more than one message. However, if receive() is called with a DatagramPacket containing a buffer of size  $n$ , and the size of the first message in the receive queue exceeds  $n$ , only the first  $n$  bytes of the message are returned. The remaining bytes are quietly discarded, with no indication to the receiving program that information has been lost!

For this reason, a receiver should always supply a DatagramPacket that has enough space to hold the largest message allowed by the application protocol at the time it calls to receive(). This technique will guarantee that no data will be lost. The maximum amount of data that can be transmitted in a DatagramPacket is 65,507 bytes—the largest payload that can be carried in a UDP datagram. Thus it's always safe to use a packet that has an array of size 65,600 or so.

It is also important to remember here that each instance of DatagramPacket has an internal notion of message length that may be changed whenever a message is received into that instance (to reflect the number of bytes in the received message). Applications that call receive() more than once with the same instance of DatagramPacket should explicitly reset the internal length to the actual buffer length before each subsequent call to receive().

Another potential source of problems for beginners is the getData() method of DatagramPacket, which always returns the entire original buffer, ignoring the internal offset and length values. Receiving a message into the DatagramPacket only modifies those locations of the buffer into which message data was placed. For example, suppose *buf* is a byte array of size 20, which has been initialized so that each byte contains its index in the array:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19
---	---	---	---	---	---	---	---	---	---	----	----	----	----	----	----	----	----	----	----

Suppose also that *dg* is a DatagramPacket, and that we set *dg*'s buffer to be the middle 10 bytes of *buf*:

```
dg.setData(buf, 5, 10);
```

Now suppose that *dgsocket* is a DatagramSocket, and that somebody sends an 8-byte message containing

41	42	43	44	45	46	47	48
----	----	----	----	----	----	----	----

to *dgsocket*. The message is received into *dg*:

```
dgsocket.receive(dg);
```

Now, calling *dg.getData()* returns a reference to the original byte array *buf*, whose contents are now

0	1	2	3	4	41	42	43	44	45	46	47	48	13	14	15	16	17	18	19
---	---	---	---	---	----	----	----	----	----	----	----	----	----	----	----	----	----	----	----

Note that only bytes 5–12 of *buf* have been modified and that, in general, the application needs to use *getOffset()* and *getData()* to access just the received data. One possibility is to copy the received data into a separate byte array, like this:

```
byte[] destBuf = new byte[dg.getLength()];
System.arraycopy(dg.getData(), dg.getOffset(), destBuf, 0, destBuf.length);
```

As of Java 1.6, we can do it in one step using the convenience method `Arrays.copyOfRange()`:

```
byte[] destBuf =  
    Arrays.copyOfRange(dg.getData(), dg.getOffset(), dg.getOffset() + dg.getLength());
```

We didn't have to do this copying in `UDPEchoServer.java` because the server did not read the data from the `DatagramPacket` at all.

### 3.2.5.6 Exercises

- 1 For `TCPEchoServer.java`, we explicitly specify the port to the socket in the constructor. We said that a socket must have a port for communication, yet we do not specify a port in `TCPEchoClient.java`. How is the echo client's socket assigned a port?
- 2 When you make a phone call, it is usually the callee that answers with "Hello." What changes to our client and server examples would be needed to implement this?
- 3 What happens if a TCP server never calls `accept()`? What happens if a TCP client sends data on a socket that has not yet been accepted at the server?
- 4 Servers are supposed to run for a long time without stopping—therefore, they must be designed to provide good service no matter what their clients do. Examine the server examples (`TCPEchoServer.java` and `UDPEchoServer.java`) and list anything you can think of that a client might do to cause it to give poor service to other clients. Suggest improvements to fix the problems that you find.
- 5 Modify `TCPEchoServer.java` to read and write only a single byte at a time, sleeping one second between each byte. Verify that `TCPEchoClient.java` requires multiple reads to successfully receive the entire echo string, even though it sent the echo string with one `write()`.
- 6 Modify `TCPEchoServer.java` to read and write a single byte and then close the socket. What happens when the `TCPEchoClient` sends a multibyte string to this server? What is happening? (Note that the response could vary by OS.)
- 7 Modify `UDPEchoServer.java` so that it only echoes every other datagram it receives. Verify that `UDPEchoClientTimeout.java` retransmits datagrams until it either receives a reply or exceeds the number of retries.
- 8 Modify `UDPEchoServer.java` so that `ECHOMAX` is much shorter (say, 5 bytes). Then use `UDPEchoClientTimeout.java` to send an echo string that is too long. What happens?
- 9 Verify experimentally the size of the largest message you can send and receive using a `DatagramPacket`.
- 10 While `UDPEchoServer.java` explicitly specifies its local port in the constructor, we do not specify the local port in `UDPEchoClientTimeout.java`. How is the UDP echo client's socket given a port number?

### 3.2.6 Sending and Receiving Encoded Data

Typically you use sockets because your program needs to provide information to, or use information provided by, another program. There is no magic: any programs that exchange information must agree on how that information will be *encoded*—represented as a sequence of bits—as well as which program sends what information when, and how the information received affects the behavior of the program. This agreement regarding the form and meaning of information exchanged over a communication channel is called a *protocol*; a protocol used in implementing a particular application is an *application nprotocol*. In our echo example from the earlier chapters, the application protocol is trivial: neither the client’s nor the server’s behavior is affected by the *contents* of the messages they exchange. Because in most real applications the behavior of clients and servers depends upon the information they exchange, application protocols are usually somewhat more complicated.

The TCP/IP protocols transport bytes of user data without examining or modifying them. This allows applications great flexibility in how they encode their information for transmission. Most application protocols are defined in terms of discrete *messages* made up of sequences of *fields*. Each field contains a specific piece of information encoded as a sequence of bits. The application protocol specifies exactly how these sequences of bits are to be arranged by the sender and interpreted, or *parsed*, by the receiver so that the latter can extract the meaning of each field. About the only constraint imposed by TCP/IP is that information must be sent and received in chunks whose length in bits is a multiple of eight. So from now on we consider messages to be sequences of *bytes*. Given this, it may be helpful to think of a transmitted message as a sequence or array of numbers, each between 0 and 255. That corresponds to the range of binary values that can be encoded in 8 bits: 00000000 for zero, 00000001 for one, 00000010 for two, and so on, up to 11111111 for 255.

When you build a program to exchange information via sockets with other programs, typically one of two situations applies: either you are designing/writing the programs on both sides of the socket, in which case you are free to define the application protocol yourself, or you are implementing a protocol that someone else has *already* specified, perhaps a protocol *standard*. In either case, the basic principles of encoding and decoding different types of information as bytes “on the wire” are the same. By the way, everything in this chapter also applies if the “wire” is a file that is written by one program and then read by another.

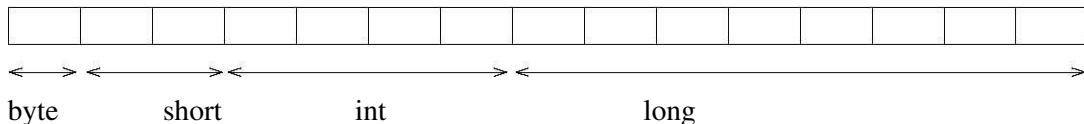
#### 3.2.6.1 Encoding Information

Let’s first consider the question of how simple values such as ints, longs, chars, and Strings can be sent and received via sockets. We have seen that bytes of information can be transmitted through a socket by writing them to an OutputStream (associated with a Socket) or encapsulating them in a DatagramPacket (which is then sent via a DatagramSocket). However, the only data types to which these operations can be applied are bytes and arrays of bytes. As a strongly typed language, Java requires that other types—int, String, and so on—be explicitly converted to byte arrays. Fortunately, the language has built-in facilities to help with such conversions. In TCPEchoClient.java, the getBytes() method of String, which converts the characters in a String instance to bytes in a standard way. Before considering the details of that kind of conversion, we first consider the representation of the most basic data types.

##### Primitive Integers

As we have already seen, TCP and UDP sockets give us the ability to send and receive sequences (arrays) of bytes, i.e., integer values in the range 0–255. Using that ability, we can encode the values of other (larger) primitive integer types. However, the sender and receiver have to agree on several things first. One is the *size*(in bytes) of each integer to be transmitted. For example, an intvalue in a Java program is represented as a 32-bit quantity. We can therefore transmit the value of any variable or constant of type intusing four bytes. Values of type short, on the other hand, are represented using 16 bits and so only require two bytes to transmit, while longs are 64 bits or eight bytes.

Let’s consider how we would encode a sequence of four integer values: a byte,a short,an int,anda long, in that order, for transmission from sender to receiver. We need a total of 15 bytes: the first contains the value of the byte, the next two contain the value of the short, the next four encode the value of the int, and the last eight bytes contain the longvalue, as shown below:



Are we ready to go? Not quite. For types that require more than one byte, we have to answer the question of

which *order* to send the bytes in. There are two obvious choices: start at the right end of the integer, with the least significant byte—so-called *little-endian* order—or at the left end, with the most significant byte—*big-endian* order. (Note that the ordering of *bits within bytes* is, fortunately, handled by the implementation in a standard way.) Consider the longvalue 123456787654321L. Its 64-bit representation (in hexadecimal) is 0x0000704885F926B1. If we transmit the bytes in big-endian order, the sequence of (decimal) byte values will look like this:

0	0	112	72	133	249	38	177
→ order of transmission							

If we transmit them in little-endian order, the sequence will be:

177	38	249	133	72	112	0	0
→ order of transmission							

The main point is that for any multibyte integer quantity, the sender and receiver need to agree on whether big-endian or little-endian order will be used. Java includes a class `ByteOrder` to denote the two possibilities. It has two static fields containing the (only) instances: `ByteOrder.BIG_ENDIAN` and `ByteOrder.LITTLE_ENDIAN`. If the sender were to use little-endian order to send the above integer, and the receiver were expecting big-endian, instead of the correct value, the receiver would interpret the transmitted eight-byte sequence as the value 12765164544669515776L.

One last detail on which the sender and receiver must agree: whether the numbers transmitted will be *signed* or *unsigned*. The four primitive integer types in Java are all signed; values are stored in *two's-complement* representation, which is the usual way of representing signed numbers. When dealing with signed  $k$ -bit numbers, the two's-complement representation of the negative integer  $-n$ ,  $1 \leq n \leq 2^{k-1}$ , is the binary value of  $2^k - n$ . The non-negative integer  $p$ ,  $0 \leq p \leq 2^{k-1} - 1$ , is encoded simply by the  $k$ -bit binary value of  $p$ . Thus, given  $k$  bits, we can represent values in the range  $-2^{k-1}$  through  $2^k - 1$  using two's-complement. Note that the most significant bit (msb) tells whether the value is positive (msb = 0) or negative (msb = 1). On the other hand, a  $k$ -bit, *unsigned* integer can encode values in the range 0 through  $2^k - 1$  directly. So for example, the 32-bit value 0xffffffff (the all-ones value) when interpreted as a signed, two's complement integer represents  $-1$ ; when interpreted as an unsigned integer, it represents 4,294,967,295. Because Java does not support unsigned integer types, encoding and decoding unsigned numbers in Java requires a little care. Assume for now that we are dealing with signed integer types.

So how do we get the correct values into the byte array of the message? To allow you to see exactly what needs to happen, here's how to do the encoding explicitly using "bit-diddling" (shifting and masking) operations. The program `BruteForceCoding.java` features a method `encodeIntBigEndian()` that can encode any value of one of the primitive types. Its arguments are the byte array into which the value is to be placed, the value to be encoded (represented as a long—which, as the largest type, can hold any of the other types), the offset in the array at which the value should start, and the size in bytes of the value to be written. If we encode at the sender, we must be able to decode at the receiver. `BruteForceCoding` also provides the `decodeIntBigEndian()` method for decoding a subset of a byte array into a Java long.

### **BruteForceCoding.java**

```

1 private static byte byteVal = 101; // one hundred and one
2 private static short shortVal = 10001; // ten thousand and one
3 private static int intValue = 100000001; // one hundred million and one
4 private static long longVal = 1000000000001L; // one trillion and one
5
6 private final static int BSIZE = Byte.SIZE / Byte.SIZE;
7 private final static int SSIZE = Short.SIZE / Byte.SIZE;
8 private final static int ISIZE = Integer.SIZE / Byte.SIZE;
9 private final static int LSIZE = Long.SIZE / Byte.SIZE;
10
11 private final static int BYTEMASK = 0xFF; // 8 bits
12
13 public static String byteArrayToDecimalString(byte[] bArray) {
14     StringBuilder rtn = new StringBuilder();
15     for (byte b : bArray) {
16         rtn.append(b & BYTEMASK).append(" ");
17     }
18     return rtn.toString();
19 }
```

```

17    }
18    return rtn.toString();
19 }
20
21 // Warning: Untested preconditions (e.g., 0 <= size <= 8)
22 public static int encodeIntBigEndian(byte[] dst, long val, int offset, int size){
23     for (int i = 0; i < size; i++) {
24         dst[offset++] = (byte) (val >> ((size - i - 1) * Byte.SIZE));
25     }
26     return offset;
27 }
28
29 // Warning: Untested preconditions (e.g., 0 <= size <= 8)
30 public static long decodeIntBigEndian(byte[] val, int offset, int size) {
31     long rtn = 0;
32     for (int i = 0; i < size; i++) {
33         rtn = (rtn << Byte.SIZE) | ((long) val[offset + i] & BYTEMASK);
34     }
35     return rtn;
36 }
37
38 public static void main(String[] args) {
39     byte[] message = new byte[BSIZE + SSIZE + ISIZE + LSIZE];
40     // Encode the fields in the target byte array
41     int offset = encodeIntBigEndian(message, byteVal, 0, BSIZE);
42     offset = encodeIntBigEndian(message, shortVal, offset, SSIZE);
43     offset = encodeIntBigEndian(message, intValue, offset, ISIZE);
44     encodeIntBigEndian(message, longVal, offset, LSIZE);
45     System.out.println("Encoded message: " + byteArrayToString(message));
46
47     // Decode several fields
48     long value = decodeIntBigEndian(message, BSIZE, SSIZE);
49     System.out.println("Decoded short = " + value);
50     value = decodeIntBigEndian(message, BSIZE + SSIZE + ISIZE, LSIZE);
51     System.out.println("Decoded long = " + value);
52
53     // Demonstrate dangers of conversion
54     offset = 4;
55     value = decodeIntBigEndian(message, offset, BSIZE);
56     System.out.println("Decoded value (offset " + offset + ", size " + BSIZE + ")=" +
57         + value);
58     byte bVal = (byte) decodeIntBigEndian(message, offset, BSIZE);
59     System.out.println("Same value as byte = " + bVal);
60 }
61
62 }

```

1 Data items to encode: lines 1–4

2 Numbers of bytes in Java integer primitives: lines 6–9

3 byteArrayToString(): lines 13–19 This method prints each byte from the given array as an unsigned decimal value. BYTEMASK keeps the byte value from being *sign-extended* when it is converted to an int in the call to append(), thus rendering it as an unsigned integer.

4 encodeIntBigEndian(): lines 22–27 The right-hand side of the assignment statement first shifts the value to the right so the byte we are interested in is in the low-order eight bits. The resulting value is then *cast* to the type byte, which throws away all but the low-order eight bits, and placed in the array at the appropriate location. This is iterated over size bytes of the given value, val. The new offset is returned so we need not keep track of it.

5 decodeIntBigEndian(): lines 30–36 Iterate over size bytes of the given array, accumulating the result in a long, which is shifted left at each iteration.

6 Demonstrate methods: lines 38–60

- Prepare array to receive series of integers: line 39
- Encode items: lines 40–44 The byte, short, int, and long are encoded into the array in the sequence described earlier.
- Print contents of encoded array: line 45
- Decode several fields from encoded byte array: lines 47–51 Output should show the decoded values equal to the original constants.
- Conversion problems: lines 53–59 At offset 4, the byte value is 245 (decimal); however, when read as a *signed* byte value, it should be -11 (recall two's-complement representation of signed integers). If we place the return value into a long, it simply becomes the last byte of a long, producing a value of 245. Placing the return value into a byte yields a value of -11. Which answer is correct depends on your application. If you expect a signed value from decoding N bytes, you must place the (long) result in a primitive integer type that uses exactly N bytes. If you expect an unsigned value from decoding N bytes, you must place the results in a primitive integer type that uses at least N+1 bytes.

Note that there are several preconditions we might consider testing at the beginning of `encodeIntBigEndian()` and `decodeIntBigEndian()`, such as  $0 \leq \text{size} \leq 8$  and `dst=null`. Can you name any others? Running the program produces output showing the following (decimal) byte values:

101	39	17	5	245	225	1	0	0	0	232	212	165	16	1
bvte	short	int				long								

As you can see, the brute-force method requires the programmer to do quite a bit of work: computing and naming the offset and size of each value, and invoking the encoding routine with the appropriate arguments. It would be even worse if the `encodeIntBigEndian()` method were not factored out as a separate method. For that reason, it is not the recommended approach, because Java provides some built-in mechanisms that are easier to use. Note that it does have the advantage that, in addition to the standard Java integer sizes, `encodeIntegerBigEndian()` works with *any* size integer from 1 to 8 bytes—for example, you can encode a seven-byte integer if you like.

A relatively easy way to construct the message in this example is to use the `DataOutputStream` and `ByteArrayOutputStream` classes. The `DataOutputStream` allows you to write primitive types like the integers we've been discussing to a stream: it provides `writeByte()`, `writeShort()`, `.writeInt()`, and `writeLong()` methods, which take an integer value and write it to the stream in the appropriately sized big-endian two's-complement representation. The `ByteArrayOutputStream` class takes the sequence of bytes written to a stream and converts it to a byte array. The code for building our message looks like this:

```
ByteArrayOutputStream          buf      =      new        ByteArrayOutputStream();
DataOutputStream            out      =      new        DataOutputStream(buf);
out.writeByte(byteVal);
out.writeShort(shortVal);
out.writeInt(intVal);
out.writeLong(longVal);
out.flush();
byte[] msg = buf.toByteArray();
```

You may want to run this code to convince yourself that it produces the same output as `BruteForceEncoding.java`.

So much for the sending side. How does the receiver recover the transmitted values? As you might expect, there are input analogues for the output facilities we used, namely `DataInputStream` and `ByteArrayInputStream`. We'll show an example of their use later, when we discuss how to parse incoming messages.

Finally, essentially everything in this subsection applies also to the `BigInteger` class, which supports arbitrarily large integers. As with the primitive integer types, sender and receiver have to agree on a specific size (number of bytes) to represent the value. However, this defeats the purpose of using a `BigInteger`, which can be arbitrarily large. One approach is to use length-based framing.

### Strings and Text

Old-fashioned *text*—strings of printable (displayable) characters—is perhaps the most common way to represent information. Text is convenient because humans are accustomed to dealing with all kinds of information represented as strings of characters in books, newspapers, and on computer displays. Thus, once we know how to encode text for transmission, we can send almost any other kind of data: first represent it as text, then encode the text. Obviously we can represent numbers and boolean values as Strings—for example "123478962", "6.02e23", "true", "false". And we've already seen that a string can be converted to a byte array by calling the `getBytes()` method (see `TCPEchoClient.java`). Alas, there is more to it than that.

To better understand what's going on, we first need to consider that text is made up of symbols or *characters*. In fact every `String` instance corresponds to a sequence (array) of *characters* (type `char[]`). A `char` value in Java is represented internally as an integer. For example, the character "a", that is, the symbol for the letter "a", corresponds to the integer 97. The character "X" corresponds to 88, and the symbol "!"(exclamation mark) corresponds to 33.

A mapping between a set of symbols and a set of integers is called a *coded character set*. You may have heard of the coded character set known as *ASCII*—American Standard Code for Information Interchange. ASCII maps the letters of the English alphabet, digits, punctuation and some other special (non-printable) symbols to integers between 0 and 127. It has been used for data transmission since the 1960s, and is used extensively in application protocols such as HTTP (the protocol used for the World Wide Web), even today. However, because it omits symbols used by many languages other than English, it is less than ideal for developing applications and protocols designed to function in today's global economy.

Java therefore uses an international standard coded character set called *Unicode* to represent values of type `char`

and String. Unicode maps symbols from most of the languages and symbol systems of the world to integers between 0 and 65,535, and is much better suited for internationalized programs. For example, the Japanese Hiragana symbol for the syllable “o” maps to the integer 12,362. Unicode includes ASCII: each symbol defined by ASCII maps to the same integer in Unicode as it does in ASCII. This provides a degree of backward compatibility between ASCII and Unicode.

So sender and receiver have to agree on a mapping from symbols to integers in order to communicate using text messages. Is that all they need to agree on? It depends. For a small set of characters with no integer value larger than 255, nothing more is needed because each character can be encoded as a single byte. For a code that may use larger integer values that require more than a single byte to represent, there is more than one way to encode those values on the wire. Thus, sender and receiver need to agree on how those integers will be represented as byte sequences—that is, an *encoding scheme*. The combination of a coded character set and a character encoding scheme is called a *charset* (see RFC 2278). It is possible to define your own charset, but there is hardly ever a reason to do so. A large number of different *standardized* charsets are in use around the world. Java provides support for the use of arbitrary charsets, and every implementation is required to support at least the following: US-ASCII (another name for ASCII), ISO-8859-1, UTF-8, UTF-16BE, UTF-16LE, UTF-16.

When you invoke the `getBytes()` method of a `String` instance, it returns a byte array containing the `String` encoded according to the *default charset* for the platform. On many platforms the default charset is UTF-8; however, in localities that make frequent use of characters outside the ASCII charset, it may be something different. To ensure that a `String` is encoded using a *particular* charset, you simply supply the name of the charset as a (`String`) argument to the `getBytes()` method. The resulting byte array contains the representation of the `String` in the given encoding.

For example, if you call `"Test!".getBytes()` on the platform on which this book was written, you get back the encoding according to UTF-8:

84	101	115	116	33
----	-----	-----	-----	----

If you call `"Test!".getBytes("UTF-16BE")`, on the other hand, you get the following array:

0	84	0	101	0	115	0	116	0	33
---	----	---	-----	---	-----	---	-----	---	----

In this case each value is encoded as a two-byte sequence, with the high-order byte first. From `"Test!".getBytes("IBM037")`, the result is:

227	133	162	163	90
-----	-----	-----	-----	----

The moral of the story is that *sender and receiver must agree on the representation for strings of text*. The easiest way for them to do that is to simply specify one of the standard charsets.

As we have seen, it is possible to write `String`s to an `OutputStream` by first converting them individually to bytes and then writing the result to the stream. That method requires that the encoding be specified on every call to `getBytes()`.

### **Bit-Diddling: Encoding Booleans**

*Bitmaps* are a very compact way to encode Boolean information, which is often used in protocols. The idea of a bitmap is that each of the bits of an integer type can encode one boolean value—typically with 0 representing false, and 1 representing true. To be able to manipulate bitmaps, you need to know how to set and clear individual bits using Java’s “bit-diddling” operations. A *mask* is an integer value that has one or more specific bits set to 1, and all others cleared (i.e., 0). We’ll deal here mainly with `int`-sized bitmaps and masks (32 bits), but everything we say applies to other integer types as well.

Let’s number the bits of a value of type `int` from 0 to 31, where bit 0 is the least significant bit. In general, the `int` value that has a 1 in bit position  $i$ , and a zero in all other bit positions, is just  $2^i$ . So bit 5 is represented by 32, bit 12 by 4096, etc. Here are some example mask declarations:

```
final int BIT5 = (1<<5);
final int BIT7 = 0x80;
final int BITS2AND3 = 12; // 8+4
int bitmap = 1234567;
```

To *set* a particular bit in an `int` variable, combine it with the mask for that bit using the bitwise-OR operation (`|`):

```
bitmap |= BIT5;
//bit5 is now one
```

To *clear* a particular bit, bitwise-AND it with the *bitwise complement* of the mask for that bit (which has ones everywhere except the particular bit, which is zero). The bitwise-AND operation in Java is `&`, while the bitwise-complement operator is `~`.

```
bitmap &= ~BIT7;
// bit 7 is now zero
```

You can set and clear multiple bits at once by OR-ing together the corresponding masks:

```
// clear bits 2, 3 and 5
bitmap &= ~(BITS2AND3|BIT5);
```

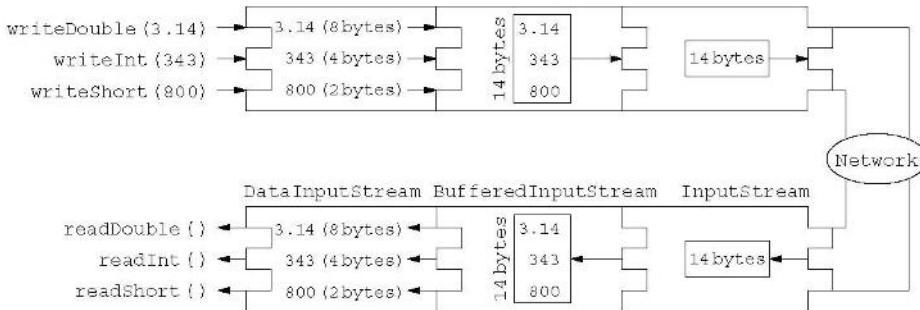
To test whether a bit is set, compare the result of the bitwise-AND of the mask and the value with zero:

```
boolean bit6Set = (bitmap & (1<<6)) != 0;
```

### 3.2.6.2 Composing I/O Streams

Java's stream classes can be composed to provide powerful capabilities. For example, we can wrap the `OutputStream` of a `Socket` instance in a `BufferedOutputStream` instance to improve performance by buffering bytes temporarily and flushing them to the underlying channel all at once. We can then wrap that instance in a `DataOutputStream` to send primitive data types. We would code this composition as follows:

```
Socket socket = new Socket(server, port);
DataOutputStream out = new DataOutputStream(
    new BufferedOutputStream(socket.getOutputStream()));
```



**Fig. 3.9. Stream composition**

**Table 1.3: Java I/O Classes**

I/O Class	Function
Buffered[Input/Output]Stream	Performs buffering for I/O optimization.
Checked[Input/Output]Stream	Maintains a checksum on data.
Cipher[Input/Output]Stream	Encrypt/Decrypt data.
Data[Input/Output]Stream	Handles read/write for primitive date types.
Digest[Input/Output]Stream	Maintains a digest on data.
GZIP[Input/Output]Stream	De/compresses a byte stream in GZIP format.
Object[Input/Output]Stream	Handles read/write objects and primitive data types.
PushbackInputStream	Allows a byte or bytes to be “unread.”
PrintOutputStream	Prints string representation of data type.
Zip[Input/Output]Stream	De/compresses a byte stream in ZIP format.

Figure 3.9 demonstrates this composition. Here, we write our primitive data values, one by one, to `DataOutputStream`, which writes the binary data to `BufferedOutputStream`, which buffers the data from the three writes and then writes once to the socket `OutputStream`, which controls writing to the network. We create a corresponding composition for the `InputStream` on the other endpoint to efficiently receive primitive data types.

A complete description of the Java I/O API is beyond the scope of this text; however, Table 1.3 provides a list of some of the relevant Java I/O classes as a starting point for exploiting its capabilities.

### 3.2.6.3 Framing and Parsing

Converting data to wire format is, of course, only half the story; the original information must be recovered at the receiver from the transmitted sequence of bytes. Application protocols typically deal with discrete messages, which are viewed as collections of fields. *Framing* refers to the problem of enabling the receiver to locate the

beginning and end of a message. Whether information is encoded as text, as multibyte binary numbers, or as some combination of the two, the application protocol must specify how the receiver of a message can determine when it has received all of the message.

Of course, if a complete message is sent as the payload of a DatagramPacket, the problem is trivial: the payload of the DatagramPacket has a definite length, and the receiver knows exactly where the message ends. For messages sent over TCP sockets, however, the situation can be more complicated because TCP has no notion of message boundaries. If the fields in a message all have fixed sizes and the message is made up of a fixed number of fields, then the size of the message is known in advance and the receiver can simply read the expected number of bytes into a byte[] buffer. This technique was used in TCPEchoClient.java, where we knew the number of bytes to expect from the server. However, when the message can vary in length—for example, if it contains some variable-length arbitrary text strings—we do not know beforehand how many bytes to read.

If a receiver tries to receive more bytes from the socket than were in the message, one of two things can happen. If no other message is in the channel, the receiver will block and be prevented from processing the message; if the sender is also blocked waiting for a reply, the result will be *deadlock*. On the other hand, if another message is in the channel, the receiver may read some or all of it as part of the first message, leading to protocol errors. Therefore framing is an important consideration when using TCP sockets.

Note that some of the same considerations apply to finding the boundaries of the individual *fields* of the message: the receiver needs to know where one ends and another begins. Thus, pretty much everything we say here about framing messages also applies to fields. However, it is simplest, and also leads to the cleanest code, if you deal with these two problems separately: first locate the end of the message, then parse the message as a whole. Here we focus on framing complete messages.

Two general techniques enable a receiver to unambiguously find the end of the message:

- *Delimiter-based*: The end of the message is indicated by a *unique marker*, an explicit byte sequence that the sender transmits immediately following the data. The marker must be known not to occur in the data.
- *Explicit length*: The variable-length field or message is preceded by a (fixed-size) length field that tells how many bytes it contains.

A special case of the delimiter-based method can be used for the last message sent on a TCP connection: the sender simply closes the sending side of the connection (using shutdownOutput() or close()) after sending the message. After the receiver reads the last byte of the message, it receives an end-of-stream indication (i.e., read() returns -1), and thus can tell that it has reached the end of the message.

The delimiter-based approach is often used with messages encoded as text: A particular character or sequence of characters is defined to mark the end of the message. The receiver simply scans the input (as characters) looking for the delimiter sequence; it returns the character string preceding the delimiter. The drawback is that *the message itself must not contain the delimiter*, otherwise the receiver will find the end of the message prematurely. With a delimiter-based framing method, the sender is responsible for ensuring that this precondition is satisfied. Fortunately so-called *stuffing* techniques allow delimiters that occur naturally in the message to be modified so the receiver will not recognize them as such; as it scans for the delimiter, it also recognizes the modified delimiters and restores them in the output message so it matches the original. The downside of such techniques is that *both* sender and receiver have to scan the message.

The length-based approach is simpler, but requires a known upper bound on the size of the message. The sender first determines the length of the message, encodes it as an integer, and prefixes the result to the message. The upper bound on the message length determines the number of bytes required to encode the length: one byte if messages always contain fewer than 256 bytes, two bytes if they are always shorter than 65,536 bytes, and so on.

In order to demonstrate these techniques, we introduce the interface Framer, which is defined below. It has two methods: frameMsg() adds framing information and outputs a given message to a given stream, while nextMsg() scans a given stream, extracting the next message.

### **Framer.java**

```
0 import java.io.IOException;
1 import java.io.OutputStream;
2
3 public interface Framer {
4     void frameMsg(byte[] message, OutputStream out) throws IOException;
5     byte[] nextMsg() throws IOException;
6 }
```

The class `DelimFramer.java` implements delimiter-based framing using the “newline” character (“\n”, byte value 10). The `frameMethod()` method does *not* do stuffing, but simply throws an exception if the byte sequence to be framed contains the delimiter. The `nextMsg()` method scans the stream until it reads the delimiter, then returns everything up to the delimiter; null is returned if the stream is empty. If some bytes of a message are accumulated and the stream ends without finding a delimiter, an exception is thrown to indicate a framing error.

### ***DelimFramer.java***

```

0 import java.io.ByteArrayOutputStream;
1 import java.io.EOFException;
2 import java.io.IOException;
3 import java.io.InputStream;
4 import java.io.OutputStream;
5
6 public class DelimFramer implements Framer {
7
8     private InputStream in; // data source
9     private static final byte DELIMITER = "\n"; // message delimiter
10
11    public DelimFramer(InputStream in) {
12        this.in = in;
13    }
14
15    public void frameMsg(byte[] message, OutputStream out) throws IOException {
16        // ensure that the message does not contain the delimiter
17        for (byte b : message) {
18            if (b == DELIMITER) {
19                throw new IOException("Message contains delimiter");
20            }
21        }
22        out.write(message);
23        out.write(DELIMITER);
24        out.flush();
25    }
26
27    public byte[] nextMsg() throws IOException {
28        ByteArrayOutputStream messageBuffer = new ByteArrayOutputStream();
29        int nextByte;
30
31        // fetch bytes until find delimiter
32        while ((nextByte = in.read()) != DELIMITER) {
33            if (nextByte == -1) { // end of stream?
34                if (messageBuffer.size() == 0) { // if no byte read
35                    return null;
36                } else { // if bytes followed by end of stream: framing error
37                    throw new EOFException("Non-empty message without delimiter");
38                }
39            }
40            messageBuffer.write(nextByte); // write byte to buffer
41        }
42
43        return messageBuffer.toByteArray();
44    }
45}

```

- 1      Constructor: lines 11–13. The input stream from which messages are to be extracted is given as an argument.
- 2      `frameMsg()` adds framing information: lines 15–25.
  - Verify well-formedness: lines 17–21. Check that the given message does not contain the delimiter; if so, throw an exception.
  - Write message: line 22. Output the framed message to the stream
  - Write delimiter: line 23.
3. `nextMsg()` extracts messages from input: lines 27–44.
  - Read each byte in the stream until the delimiter is found: line 32.
  - Handle end of stream: lines 33–39. If the end of stream occurs before finding the delimiter, throw an exception if any bytes have been read since construction of the framer or the last delimiter; otherwise return null to indicate that all messages have been received.
  - Write non-delimiter byte to message buffer: line 40.
  - Return contents of message buffer as byte array: line 43.

There’s a limitation to our delimiting framer: it does not support multibyte delimiters. We leave fixing this as an

exercise for the reader.

The class LengthFramer.java implements length-based framing for messages up to 65,535 ( $2^{16} - 1$ ) bytes in length. The sender determines the length of the given message and writes it to the output stream as a two-byte, big-endian integer, followed by the complete message. On the receiving side, we use a DataInputStream to be able to read the length as an integer; the readFully() method blocks until the given array is completely full, which is exactly what we need here. Note that, with this framing method, the sender does not have to inspect the content of the message being framed; it needs only to check that the message does not exceed the length limit.

### LengthFramer.java

```
0 import java.io.DataInputStream;
1 import java.io.EOFException;
2 import java.io.IOException;
3 import java.io.InputStream;
4 import java.io.OutputStream;
5
6 public class LengthFramer implements Framer {
7     public static final int MAXMESSAGELENGTH = 65535;
8     public static final int BYTEMASK = 0xff;
9     public static final int SHORTMASK = 0xffff;
10    public static final int BYTESHIFT = 8;
11
12    private DataInputStream in; // wrapper for data I/O
13
14    public LengthFramer(InputStream in) throws IOException {
15        this.in = new DataInputStream(in);
16    }
17
18    public void frameMsg(byte[] message, OutputStream out) throws IOException {
19        if (message.length > MAXMESSAGELENGTH) {
20            throw new IOException("message too long");
21        }
22        // write length prefix
23        out.write((message.length >> BYTESHIFT) & BYTEMASK);
24        out.write(message.length & BYTEMASK);
25        // write message
26        out.write(message);
27        out.flush();
28    }
29
30    public byte[] nextMsg() throws IOException {
31        int length;
32        try {
33            length = in.readUnsignedShort(); // read 2 bytes
34        } catch (EOFException e) { // no (or 1 byte) message
35            return null;
36        }
37        // 0 <= length <= 65535
38        byte[] msg = new byte[length];
39        in.readFully(msg); // if exception, it's a framing error.
40        return msg;
41    }
42 }
```

1 Constructor: lines 14–16. Take the input stream source for framed messages and wrap it in a DataInputStream.  
2 frameMsg() adds framing information: lines 18–28.

- Verify length: lines 19–21. Because we use a two-byte length field, the length cannot exceed 65,535. (Note that this value is too big to store in a short, so we write it a byte at a time.)
  - Output length field: lines 23–24. Output the message bytes prefixed by the length (unsigned short).
  - Output message: line 26
3. nextMsg() extracts next frame from input: lines 30–41.
- Read the prefix length: lines 32–36. The readUnsignedShort() method reads two bytes, interprets them as a big-endian integer, and returns their value as an int.
  - Read the specified number of bytes: lines 38–39. The readFully() method blocks until enough bytes to fill the given array have been returned.
  - Return bytes as message: line 40

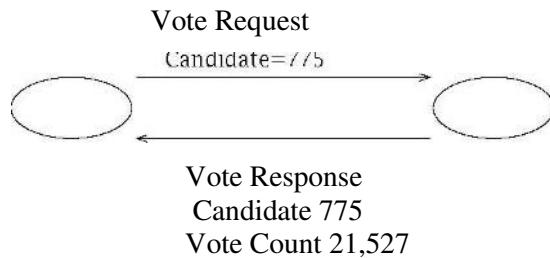
### 3.2.6.4 Java-Specific Encodings

When you use sockets, generally either you are building the programs on both ends of the communication channel—in which case you also have complete control over the protocol—or you are communicating using a *given* protocol, which you have to implement. When you know that (i) both ends of the communication will be implemented in Java, and (ii) you have complete control over the protocol, you can make use of Java’s built-in facilities like the *Serializable* interface or the *Remote Method Invocation (RMI)* facility. RMI lets you invoke methods on different Java virtual machines, hiding all the messy details of argument encoding and decoding. Serialization handles conversion of actual Java objects to byte sequences for you, so you can transfer actual instances of Java objects between virtual machines.

These capabilities are not always the best solution, for several reasons. First, because they are very general facilities, they are not the most efficient in terms of communication overhead. For example, the serialized form of an object generally includes information that is meaningless outside the context of the Java Virtual Machine (JVM). Second, *Serializable* and *Externalizable* cannot be used when a different wire format has already been specified—for example, by a standardized protocol. And finally, custom-designed classes have to provide their own implementations of the serialization interfaces, *and this is not easy to get right*. Again, there are certainly situations where these built-in facilities are useful; but sometimes it is simpler, easier, or more efficient to “roll your own.”

### 3.3.6.5 Constructing and Parsing Protocol Messages

We close this chapter with a simple example to illustrate some techniques you might use to implement a protocol specified by someone else. The example is a simple “voting” protocol as shown in Figure 3.10.



**Fig. 3.10. Voting protocol.**

Here a client sends a request message to a server; the message contains a candidate ID, which is an integer between 0 and 1000. Two types of requests are supported. An *inquiry* asks the server how many votes have been cast for the given candidate. The server sends back a response message containing the original candidate ID and the vote total (as of the time the request was received) for that candidate. A *voting* request actually casts a vote for the indicated candidate. The server again responds with a message containing the candidate ID and the vote total.

In implementing a protocol, it is helpful to define a class to contain the information carried in a message. The class provides methods for manipulating the fields of the message—while maintaining the invariants that are supposed to hold among those fields. For our simple example, the messages sent by client and server are very similar. The only difference is that the messages sent by the server contain the vote count and a flag indicating that they are responses (not requests). In this case, we can get away with a single class for both kinds of messages. The *VoteMsg.java* class shows the basic information in each message:

- a boolean *isInquiry*, which is true if the requested transaction is an inquiry (and false if it is an actual vote);
- a boolean *isResponse* indicating whether the message is a response (sent by the server) or request;
- an integer *candidateID* that identifies the candidate;
- a long *voteCount* indicating the vote total for the requested candidate

The class maintains the following invariants among the fields:

1. *candidateID* is in the range 0–1000.
2. *voteCount* is only nonzero in response messages (*isResponse* is true).
3. *voteCount* is non-negative.

#### **VoteMsg.java**

```
0 public class VoteMsg {  
1 private boolean isInquiry; // true if inquiry; false if vote  
2 private boolean isResponse; // true if response from server
```

```

3 private int candidateID; // in [0,1000]
4 private long voteCount; // nonzero only in response
5
6 public static final int MAX_CANDIDATE_ID = 1000;
7
8 public VoteMsg(boolean isResponse,boolean isInquiry,int candidateID,longvoteCount)
9     throws IllegalArgumentException {
10    // check invariants
11    if (voteCount != 0 && !isResponse) {
12        throw new IllegalArgumentException("Request vote count must be zero");
13    }
14    if (candidateID < 0 || candidateID > MAX_CANDIDATE_ID) {
15        throw new IllegalArgumentException("Bad Candidate ID: " + candidateID);
16    }
17    if (voteCount < 0) {
18        throw new IllegalArgumentException("Total must be >= zero");
19    }
20    this.candidateID = candidateID;
21    this.isResponse = isResponse;
22    this.isInquiry = isInquiry;
23    this.voteCount = voteCount;
24 }
25
26 public void setInquiry(boolean isInquiry) {
27    this.isInquiry = isInquiry;
28 }
29
30 public void setResponse(boolean isResponse) {
31    this.isResponse = isResponse;
32 }
33
34 public boolean isInquiry() {
35    return isInquiry;
36 }
37
38 public boolean isResponse() {
39    return isResponse;
40 }
41
42 public void setCandidateID(int candidateID) throws IllegalArgumentException {
43    if (candidateID < 0 || candidateID > MAX_CANDIDATE_ID) {
44        throw new IllegalArgumentException("Bad Candidate ID: " + candidateID);
45    }
46    this.candidateID = candidateID;
47 }
48
49 public int getCandidateID() {
50    return candidateID;
51 }
52
53 public void setVoteCount(long count) {
54    if ((count != 0 && !isResponse) || count < 0) {
55        throw new IllegalArgumentException("Bad vote count");
56    }
57    voteCount = count;
58 }
59
60 public long getVoteCount() {
61    return voteCount;
62 }
63
64 public String toString() {
65    String res = (isInquiry ? "inquiry":"vote") + " for candidate " + candidateID;
66    if (isResponse) {
67        res = "response to " + res + " who now has " + voteCount + " vote(s)";
68    }

```

```

69     return res;
70 }
71 }
```

Now that we have a Java representation of a vote message, we need some way to encode and decode according to some protocol. A `VoteMsgCoder` provides the methods for vote message serialization and deserialization.

### ***VoteMsgCoder.java***

```

0 import java.io.IOException;
1
2 public interface VoteMsgCoder {
3     byte[] toWire(VoteMsg msg) throws IOException;
4     VoteMsg fromWire(byte[] input) throws IOException;
5 }
```

The `toWire()` method converts the vote message to a sequence of bytes according to a particular protocol, and the `fromWire()` method parses a given sequence of bytes according to the same protocol and constructs an instance of the message class.

To illustrate the different methods of encoding information, we present two implementations of `VoteMsgCoder`, one using a text-based encoding and one using a binary encoding. If you were guaranteed a single encoding that would never change, the `toWire()` and `fromWire()` methods could be specified as part of `VoteMsg`. Our purpose here is to emphasize that the abstract representation is independent of the details of the encoding.

### ***Text-Based Representation***

We first present a version in which messages are encoded as text. The protocol specifies that the text be encoded using the US-ASCII charset. The message begins with a so-called “magic string”—a sequence of characters that allows a recipient to quickly recognize the message as a Voting protocol message, as opposed to random garbage that happened to arrive over the network. The `Vote/Inquiry` boolean is encoded with the character ‘v’ for a vote or ‘i’ for an inquiry. The message’s status as a response is indicated by the presence of the character ‘R’. Then comes the candidate ID, followed by the vote count, both encoded as decimal strings. The `VoteMsgTextCoder` provides a text-based encoding of `VoteMsg`.

### ***VoteMsgTextCoder.java***

```

0 import java.io.ByteArrayInputStream;
1 import java.io.IOException;
2 import java.io.InputStreamReader;
3 import java.util.Scanner;
4
5 public class VoteMsgTextCoder implements VoteMsgCoder {
6 /*
7 * Wire Format "VOTEPROTO" <"v"|"i"> [<RESPFLAG>] <CANDIDATE> [<VOTECNT>]
8 * Charset is fixed by the wire format.
9 */
10
11 // Manifest constants for encoding
12 public static final String MAGIC = "Voting";
13 public static final String VOTESTR = "v";
14 public static final String INQSTR = "i";
15 public static final String RESPONSESTR = "R";
16
17 public static final String CHARSETNAME = "US-ASCII";
18 public static final String DELIMSTR = " ";
19 public static final int MAX_WIRE_LENGTH = 2000;
20
21 public byte[] toWire(VoteMsg msg) throws IOException {
22     String msgString = MAGIC + DELIMSTR + (msg.isInquiry() ? INQSTR : VOTESTR)
23             + DELIMSTR + (msg.isResponse() ? RESPONSESTR + DELIMSTR : "") +
24             + Integer.toString(msg.getCandidateID()) + DELIMSTR
25             + Long.toString(msg.getVoteCount());
26     byte data[] = msgString.getBytes(CHARSETNAME);
27     return data;
28 }
29
30 public VoteMsg fromWire(byte[] message) throws IOException {
```

```

31  ByteArrayInputStream msgStream = new ByteArrayInputStream(message);
32  Scanner s = new Scanner(new InputStreamReader(msgStream, CHARSETNAME));
33  boolean isInquiry;
34  boolean isResponse;
35  int candidateID;
36  long voteCount;
37  String token;
38
39 try {
40     token = s.next();
41     if (!token.equals(MAGIC)) {
42         throw new IOException("Bad magic string: " + token);
43     }
44     token = s.next();
45     if (token.equals(VOTESTR)) {
46         isInquiry = false;
47     } else if (!token.equals(INQSTR)) {
48         throw new IOException("Bad vote/inq indicator: " + token);
49     } else {
50         isInquiry = true;
51     }
52
53     token = s.next();
54     if (token.equals(RESPONSESTR)) {
55         isResponse = true;
56         token = s.next();
57     } else {
58         isResponse = false;
59     }
60     // Current token is candidateID
61     // Note: isResponse now valid
62     candidateID = Integer.parseInt(token);
63     if (isResponse) {
64         token = s.next();
65         voteCount = Long.parseLong(token);
66     } else {
67         voteCount = 0;
68     }
69 } catch (IOException ioe) {
70     throw new IOException("Parse error...");
71 }
72 return new VoteMsg(isResponse, isInquiry, candidateID, voteCount);
73 }
74 }
```

The `toWire()` method simply constructs a string containing all the fields of the message, separated by white space. The `fromWire()` method first looks for the “Magic” string; if it is not the first thing in the message, it throws an exception. This illustrates a very important point about implementing protocols: *never assume anything about any input from the network.*

Your program must always be prepared for any possible inputs, and handle them gracefully. In this case, the `fromWire()` method throws an exception if the expected string is not present. Otherwise, it gets the fields token by token, using the `Scanner` instance. Note that the number of fields in the message depends on whether it is a request (sent by the client) or response (sent by the server). `fromWire()` throws an exception if the input ends prematurely or is otherwise malformed.

### **Binary Representation**

Next we present a different way to encode the Voting protocol message. In contrast with the text-based format, the binary format uses fixed-size messages. Each message begins with a one-byte field that contains the “magic” value 010101 in its high-order six bits. This little bit of redundancy provides the receiver with a small degree of assurance that it is receiving a proper voting message. The two low-order bits of the first byte encode the two booleans. The second byte of the message always contains zeros, and the third and fourth bytes contain the `candidateID`. The final eight bytes of a response message (only) contain the `vote count`.

### ***VoteMsgBinCoder.java***

```

0 import java.io.ByteArrayInputStream;
1 import java.io.ByteArrayOutputStream;
2 import java.io.DataInputStream;
3 import java.io.DataOutputStream;
4 import java.io.IOException;
5
6 /* Wire Format
7 * 1 1 1 1 1 1
8 * 0 1 2 3 4 5 6 7 8 9 0 1 2 3 4 5
9 * +---+---+---+---+---+---+---+---+---+---+
10 * | Magic |Flags| ZERO |
11 * +---+---+---+---+---+---+---+---+---+---+
12 * | Candidate ID |
13 * +---+---+---+---+---+---+---+---+---+---+
14 * | |
15 * | Vote Count (only in response) |
16 * | |
17 * | |
18 * +---+---+---+---+---+---+---+---+---+---+
19 */
20 public class VoteMsgBinCoder implements VoteMsgCoder {
21
22 // manifest constants for encoding
23 public static final int MIN_WIRE_LENGTH = 4;
24 public static final int MAX_WIRE_LENGTH = 1
25 public static final int MAGIC_MASK = 0xfc00;
26 public static final int MAGIC_SHIFT = 8;
27 public static final int RESPONSE_FLAG = 0x0200;
28 public static final int INQUIRE_FLAG = 0x0100;
29
30
31 public byte[] toWire(VoteMsg msg) throws IOException {
32     ByteArrayOutputStream byteStream = new ByteArrayOutputStream();
33     DataOutputStream out = new DataOutputStream(byteStream); // converts ints
34
35     short magicAndFlags = MAGIC;
36     if (msg.isInquiry()) {
37         magicAndFlags |= INQUIRE_FLAG;
38     }
39     if (msg.isResponse()) {
40         magicAndFlags |= RESPONSE_FLAG;
41     }
42     out.writeShort(magicAndFlags);
43     // We know the candidate ID will fit in a short: it's > 0 && < 1000
44     out.writeShort((short) msg.getCandidateID());
45     if (msg.isResponse()) {
46         out.writeLong(msg.getVoteCount());
47     }
48     out.flush();
49     byte[] data = byteStream.toByteArray();
50     return data;
51 }
52
53 public VoteMsg fromWire(byte[] input) throws IOException {
54     // sanity checks
55     if (input.length < MIN_WIRE_LENGTH) {
56         throw new IOException("Runt message");
57     }
58     ByteArrayInputStream bs = new ByteArrayInputStream(input);
59     DataInputStream in = new DataInputStream(bs);
60     int magic = in.readShort();
61     if ((magic & MAGIC_MASK) != MAGIC) {
62         throw new IOException("Bad Magic #: " +
63                             ((magic & MAGIC_MASK) >> MAGIC_SHIFT));
64     }
65     boolean resp = ((magic & RESPONSE_FLAG) != 0);
66     boolean inq = ((magic & INQUIRE_FLAG) != 0);

```

```

67     int candidateID = in.readShort();
68     if (candidateID < 0 || candidateID > 1000) {
69         throw new IOException("Bad candidate ID: " + candidateID);
70     }
71     long count = 0;
72     if (resp) {
73         count = in.readLong();
74         if (count < 0) {
75             throw new IOException("Bad vote count: " + count);
76         }
77     }
78     // Ignore any extra bytes
79     return new VoteMsg(resp, inq, candidateID, count);
80 }
81 }
```

We create a `ByteArrayOutputStream` and wrap it in a `DataOutputStream` to receive the result. The encoding method takes advantage of the fact that the high-order two bytes of a valid candidateID are always zero. Note also the use of bitwise-or operations to encode the booleans using a single bit each.

### 3.2.6.6 Sending and Receiving over the stream

Sending a message over a stream is as simple as creating it, calling `toWire()`, adding appropriate framing information, and writing it. Receiving, of course, does things in the opposite order. This approach applies to TCP; in UDP explicit framing is not necessary, because message boundaries are preserved. To demonstrate this, consider a vote server that 1) maintains a mapping of candidate IDs to number of votes, 2) counts submitted votes, and 3) responds to inquiries and votes with the current count for the specified candidate. We begin by implementing a service for use by vote servers. When a vote server receives a vote message, it handles the request by calling the `handleRequest()` method of `VoteService`.

#### `VoteService.java`

```

0 import java.util.HashMap;
1 import java.util.Map;
2
3 public class VoteService {
4
5     // Map of candidates to number of votes
6     private Map<Integer, Long> results = new HashMap<Integer, Long>();
7
8     public VoteMsg handleRequest(VoteMsg msg) {
9         if (msg.isResponse()) { // If response, just send it back
10             return msg;
11         }
12         msg.setResponse(true); // Make message a response
13         // Get candidate ID and vote count
14         int candidate = msg.getCandidateID();
15         Long count = results.get(candidate);
16         if (count == null) {
17             count = 0L; // Candidate does not exist
18         }
19         if (!msg.isInquiry()) {
20             results.put(candidate, ++count); // If vote, increment count
21         }
22         msg.setVoteCount(count);
23         return msg;
24     }
25 }
```

1 Create map of candidate ID to vote count: line 6 For inquiries, the given candidate ID is used to look up the candidate's vote count in the map. For votes, the incremented vote count is stored back in the map.

2 `handleRequest()`: lines 8–24

- Return a response: lines 9–12 If the vote message is already a response, we send it back without processing or modification. Otherwise we set the response flag.
- Find current vote count: lines 13–18 Find the candidate by ID in the map and fetch the vote count. If the candidate ID does not already exist in the map, set the count to 0.
- Update count, if vote: lines 19–21 If the candidate did not previously exist, this creates a new mapping; otherwise, it simply modifies

- an existing mapping.
- Set vote count and return message: lines 22–23

Next we show how to implement a TCP voting client that connects over a TCP socket to the voting server, sends an inquiry followed by a vote, and then receives the inquiry and vote responses.

### **VoteClientTCP.java**

```

0 import java.io.OutputStream;
1 import java.net.Socket;
2
3 public class VoteClientTCP {
4
5     public static final int CANDIDATEID = 888;
6
7     public static void main(String args[]) throws Exception {
8
9         if (args.length != 2) { // Test for correct # of args
10             throw new IllegalArgumentException("Parameter(s): <Server> <Port>");
11         }
12
13         String destAddr = args[0]; // Destination address
14         int destPort = Integer.parseInt(args[1]); // Destination port
15
16         Socket sock = new Socket(destAddr, destPort);
17         OutputStream out = sock.getOutputStream();
18
19         // Change Bin to Text for a different framing strategy
20         VoteMsgCoder coder = new VoteMsgBinCoder();
21         // Change Length to Delim for a different encoding strategy
22         Framer framer = new LengthFramer(sock.getInputStream());
23
24         // Create an inquiry request (2nd arg = true)
25         VoteMsg msg = new VoteMsg(false, true, CANDIDATEID, 0);
26         byte[] encodedMsg = coder.toWire(msg);
27
28         // Send request
29         System.out.println("Sending Inquiry (" + encodedMsg.length + " bytes): ");
30         System.out.println(msg);
31         framer.frameMsg(encodedMsg, out);
32
33         // Now send a vote
34         msg.setInquiry(false);
35         encodedMsg = coder.toWire(msg);
36         System.out.println("Sending Vote (" + encodedMsg.length + " bytes): ");
37         framer.frameMsg(encodedMsg, out);
38
39         // Receive inquiry response
40         encodedMsg = framer.nextMsg();
41         msg = coder.fromWire(encodedMsg);
42         System.out.println("Received Response (" + encodedMsg.length
43                         + " bytes): ");
44         System.out.println(msg);
45
46         // Receive vote response
47         msg = coder.fromWire(framer.nextMsg());
48         System.out.println("Received Response (" + encodedMsg.length
49                         + " bytes): ");
50         System.out.println(msg);
51
52         sock.close();
53     }
54 }
```

- 1 Process arguments: lines 9–14
- 2 Create socket, get output stream: lines 16–17
- 3 Create binary coder and length-based framer: lines 20–22 We will encode/decode our vote messages using a coder. We elect to use a binary encoder for our protocol. Next, since TCP is a stream-based service, we need to provide our own framing. Here we use the

LengthFramer, which prefixes each message with a length. Note that we could easily switch to using delimiter-based framing and/or text encoding simply by changing the concrete types of our VoteMsgCoder and Framer to VoteMsgTextCoder and DelimFramer, respectively.

4 Create and send messages: lines 24–37 Create, encode, frame and send an inquiry, followed by a vote message for the same candidate.

5 Get and parse responses: lines 39–50 The nextMsg() method returns the contents of the next encoded message, which we parse/decode via fromWire().

6 Close socket: line 52

Next we demonstrate the TCP version of the vote server. Here the server repeatedly accepts a new client connection and uses the VoteService to generate responses to the client vote messages.

### **VoteServerTCP.java**

```
0 import java.io.IOException;
1 import java.net.ServerSocket;
2 import java.net.Socket;
3
4 public class VoteServerTCP {
5
6     public static void main(String args[]) throws Exception {
7
8         if (args.length != 1) { // Test for correct # of args
9             throw new IllegalArgumentException("Parameter(s): <Port>");
10        }
11
12        int port = Integer.parseInt(args[0]); // Receiving Port
13
14        ServerSocket servSock = new ServerSocket(port);
15        // Change Bin to Text on both client and server for different encoding
16        VoteMsgCoder coder = new VoteMsgBinCoder();
17        VoteService service = new VoteService();
18
19        while (true) {
20            Socket clntSock = servSock.accept();
21            System.out.println("Handling client at " + clntSock.getRemoteSocketAddress());
22            // Change Length to Delim for a different framing strategy
23            Framer framer = new LengthFramer(clntSock.getInputStream());
24            try {
25                byte[] req;
26                while ((req = framer.nextMsg()) != null) {
27                    System.out.println("Received message (" + req.length + " bytes)");
28                    VoteMsg responseMsg = service.handleRequest(coder.fromWire(req));
29                    framer.frameMsg(coder.toWire(responseMsg), clntSock.getOutputStream());
30                }
31            } catch (IOException ioe) {
32                System.err.println("Error handling client: " + ioe.getMessage());
33            } finally {
34                System.out.println("Closing connection");
35                clntSock.close();
36            }
37        }
38    }
39 }
```

1 Establish coder and vote service for server: lines 15–17

2. Repeatedly accept and handle client connections: lines 19–37

- Accept new client, print address: lines 20–21

- Create framer for client: line 23

- Fetch and decode messages from client: lines 26–28 Repeatedly request next message from framer until it returns null, indicating an end of messages.

- Process message, send response: lines 28–29 Pass the decoded message to the voting service for handling. Encode, frame, and send the returned response message.

The UDP voting client works very similarly to the TCP version. Note that for UDP we don't need to use a framer because UDP maintains message boundaries for us. For UDP, we use the text encoding for our messages; however, this can be easily changed, as long as client and server agree.

### **VoteClientUDP.java**

```
0 import java.io.IOException;
1 import java.net.DatagramPacket;
2 import java.net.DatagramSocket;
3 import java.net.InetAddress;
4 import java.util.Arrays;
5
6 public class VoteClientUDP {
7
8     public static void main(String args[]) throws IOException {
9
10        if (args.length != 3) { // Test for correct # of args
11            throw new IllegalArgumentException("Parameter(s): <Destination>" +
12                  " <Port> <Candidate#>");
13        }
14
15        InetAddress destAddr = InetAddress.getByName(args[0]); // Destination addr
16        int destPort = Integer.parseInt(args[1]); // Destination port
17        int candidate = Integer.parseInt(args[2]); // 0 <= candidate <= 1000 req'd
18
19        DatagramSocket sock = new DatagramSocket(); // UDP socket for sending
20        sock.connect(destAddr, destPort);
21
22        // Create a voting message (2nd param false = vote)
23        VoteMsg vote = new VoteMsg(false, false, candidate, 0);
24
25        // Change Text to Bin here for a different coding strategy
26        VoteMsgCoder coder = new VoteMsgTextCoder();
27
28        // Send request
29        byte[] encodedVote = coder.toWire(vote);
30        System.out.println("Sending Text-Encoded Request (" + encodedVote.length
31                           + " bytes): ");
32        System.out.println(vote);
33        DatagramPacket message = new DatagramPacket(encodedVote, encodedVote.length);
34        sock.send(message);
35
36        // Receive response
37        message = new DatagramPacket(new byte[VoteMsgTextCoder.MAX_WIRE_LENGTH],
38                                     VoteMsgTextCoder.MAX_WIRE_LENGTH);
39        sock.receive(message);
40        encodedVote = Arrays.copyOfRange(message.getData(), 0, message.getLength());
41
42        System.out.println("Received Text-Encoded Response (" + encodedVote.length
43                           + " bytes): ");
44        vote = coder.fromWire(encodedVote);
45        System.out.println(vote);
46    }
47 }
```

1       Setup DatagramSocket and connect: lines 10–20 By calling connect(), we don't have to 1) specify a remote address/port for each datagram we send and 2) test the source of any datagrams we receive.

2       Create vote and coder: lines 22–26 This time we use a text coder; however, we could easily change to a binary coder. Note that we don't need a framer because UDP already preserves message boundaries for us as long as each send contains exactly one vote message.

3       Send request to the server: lines 28–34

4       Receive, decode, and print server response: lines 36–45 When creating the DatagramPacket, we need to know the maximum message size to avoid datagram truncation. Of course, when we decode the datagram, we only use the actual bytes from the datagram so we use Arrays.copyOfRange() to copy the subsequence of the datagram backing array.

Finally, here is the UDP voting server, which, again, is very similar to the TCP version.

### **VoteServerUDP.java**

```
0 import java.io.IOException;
1 import java.net.DatagramPacket;
2 import java.net.DatagramSocket;
3 import java.util.Arrays;
```

```

4
5 public class VoteServerUDP {
6
7     public static void main(String[] args) throws IOException {
8
9         if (args.length != 1) { // Test for correct # of args
10             throw new IllegalArgumentException("Parameter(s): <Port>");
11         }
12
13     int port = Integer.parseInt(args[0]); // Receiving Port
14
15     DatagramSocket sock = new DatagramSocket(port); // Receive socket
16
17     byte[] inBuffer = new byte[VoteMsgTextCoder.MAX_WIRE_LENGTH];
18     // Change Bin to Text for a different coding approach
19     VoteMsgCoder coder = new VoteMsgTextCoder();
20     VoteService service = new VoteService();
21
22     while (true) {
23         DatagramPacket packet = new DatagramPacket(inBuffer, inBuffer.length);
24         sock.receive(packet);
25         byte[] encodedMsg=Arrays.copyOfRange(packet.getData(),0,packet.getLength());
26         System.out.println("Handling request from "+packet.getSocketAddress()+" ("+
27                             + encodedMsg.length + " bytes)");
28
29         try {
30             VoteMsg msg = coder.fromWire(encodedMsg);
31             msg = service.handleRequest(msg);
32             packet.setData(coder.toWire(msg));
33             System.out.println("Sending response (" + packet.getLength() + " bytes):");
34             System.out.println(msg);
35             sock.send(packet);
36         } catch (IOException ioe) {
37             System.err.println("Parse error in message: " + ioe.getMessage());
38         }
39     }
40 }
41 }
```

1      Setup: lines 17–20 Create reception buffer, coder, and vote service for server.

2      Repeatedly accept and handle client vote messages: lines 22–39

- Set up DatagramPacketto receive: line 23Reset the data area to the input buffer on each iteration.
- Receive datagram, extract data: lines 24–25UDP does the framing for us!
- Decode and handle request: lines 30–31The service returns a response to the message.
- Encode and send response message: lines 32–35

### 3.2.6.7 Exercises

1      Positive integers larger than  $2^{31} - 1$  (and less than  $2^{32} - 1$ ) cannot be represented as ints in Java, yet they can be represented as 32-bit binary numbers. Write a method to write such an integer to a stream. It should take a long and an OutputStream as parameters.

2      Extend the DelimFramer class to handle arbitrary multiple-byte delimiters. Be sure your implementation is efficient.

3      Assuming that all byte values are equally likely, what is the probability that a message consisting of random bits will pass the “magic test” in VoteMsgBin? Suppose an ASCII-encoded text message is sent to a program expecting a binary-encoded voteMsg. Which characters would enable the message to pass the “magic test” if they are the first in the message?

4      The encodeIntBigEndian() method of BruteForceEncoding only works if several preconditions are met such as  $0 \leq size \leq 8$ . Modify the method to test for these preconditions and throw an exception if any are violated.

### 3.2.7 Multitasking

Our basic TCP echo server handles one client at a time. If a client connects while another is already being serviced, the server will not echo the new client's data until it has finished with the current client, although the new client will be able to send data as soon as it connects. This type of server is known as an *iterative server*. Iterative servers handle clients sequentially, finishing with one client before servicing the next. They work best for applications where each client requires a small, bounded amount of server connection time; however, if the time to handle a client can be long, the wait experienced by subsequent clients may be unacceptable.

To demonstrate the problem, add a 10-second sleep using Thread.sleep() after the Socket constructor call in TCPEchoClient.java and experiment with several clients simultaneously accessing the TCP echo server. Here the sleep call simulates an operation that takes significant time, such as slow file or network I/O. Note that a new client must wait for all already-connected clients to complete before it gets service.

What we need is some way for each connection to proceed independently, without interfering with other connections. Java *threads* provide exactly that: a convenient mechanism allowing servers to handle many clients simultaneously. Using threads, a single application can work on several tasks concurrently, as if multiple copies of the Java Virtual Machine were running. In our echo server, we can give responsibility for each client to an independently executing thread. All of the examples we have seen so far consist of a single thread, which simply executes the main() method.

The two approaches to coding *concurrent servers* are *thread-per-client*, where a new thread is spawned to handle each client connection, and *thread pool*, where connections are assigned to a pre spawned set of threads. We shall also describe the built-in Java facilities that simplify the use of these strategies for multithreaded servers.

#### 3.2.7.1 Mutithreaded server

Since the multitasking server approaches we are going to describe are independent of the particular client-server protocol, we want to be able to use the same protocol implementation for both approaches for concurrent servers. The code for the echo protocol is given in the class EchoProtocol. This class encapsulates the per-client processing in the static method handleEchoClient(). This code is almost identical to the connection-handling portion of TCPEchoServer.java, except that a logging capability has been added; the method takes references to the client Socket and the Logger instance as arguments.

The class implements Runnable (the run() method simply invokes handleEchoClient() with the instance's Socket and Logger references), so we can create a thread that independently executes run(). Alternatively, the server-side protocol processing can be invoked by calling the static method directly (passing it the Socket and Logger references).

##### **EchoProtocol.java**

```
0 import java.io.IOException;
1 import java.io.InputStream;
2 import java.io.OutputStream;
3 import java.net.Socket;
4 import java.util.logging.Level;
5 import java.util.logging.Logger;
6
7 public class EchoProtocol implements Runnable {
8     private static final int BUFSIZE = 32; // Size (in bytes) of I/O buffer
9     private Socket clntSock; // Socket connect to client
10    private Logger logger; // Server logger
11
12    public EchoProtocol(Socket clntSock, Logger logger) {
13        this.clntSock = clntSock;
14        this.logger = logger;
15    }
16
17    public static void handleEchoClient(Socket clntSock, Logger logger) {
18        try {
19            // Get the input and output I/O streams from socket
20            InputStream in = clntSock.getInputStream();
21            OutputStream out = clntSock.getOutputStream();
22
23            int recvMsgSize; // Size of received message
```

```

24     int totalBytesEchoed = 0; // Bytes received from client
25     byte[] echoBuffer = new byte[BUFSIZE]; // Receive Buffer
26     // Receive until client closes connection, indicated by -1
27     while ((recvMsgSize = in.read(echoBuffer)) != -1) {
28         out.write(echoBuffer, 0, recvMsgSize);
29         totalBytesEchoed += recvMsgSize;
30     }
31
32     logger.info("Client " + clntSock.getRemoteSocketAddress() + ", echoed "
33                 + totalBytesEchoed + " bytes.");
34
35 } catch (IOException ex) {
36     logger.log(Level.WARNING, "Exception in echo protocol", ex);
37 } finally {
38     try {
39         clntSock.close();
40     } catch (IOException e) {
41     }
42 }
43 }
44
45 public void run() {
46     handleEchoClient(clntSock, logger);
47 }
48 }
```

1 Declaration of implementation of the Runnableinterface: line 7

2 Member variables and constructor: lines 8–15 Each instance of EchoProtocolcontains a socket for the connection and a reference to the logger instance.

3. handleEchoClient(): lines 17–43Implement the echo protocol:

- Get the input/output streams from the socket: lines 20–21
- Receive and echo: lines 25–30 Loop until the connection is closed (as indicated by read()returning –1), writing whatever is received back immediately.
- Record the connection details in the log: lines 32–33 Record the SocketAddressof the remote end along with the number of bytes echoed.
- Handle exceptions: line 36 Log any exceptions.

Your server is up and running with thousands of clients per minute. Now a user reports a problem. How do you determine what happened? Is the problem at your server? Perhaps the client is violating the protocol. To deal with this scenario, most servers log their activities. This practice is so common that Java now includes built-in logging facilities in the `java.util.logging` package. We provide a very basic introduction to logging here; however, be aware that there are many more features to enterprise-level logging.

We begin with the `Logger` class, which represents a logging facility that may be local or remote. Through an instance of this class, we can record the various server activities as shown in `EchoProtocol`. You may use several loggers in your server, each serving a different purpose and potentially behaving in a different way. For example, you may have separate loggers for operations, security, and error messages. In Java each logger is identified by a globally unique name. To get an instance of `Logger`, call the static factory method `Logger.getLogger()`asfollows:

```
Logger logger = Logger.getLogger("practical");
```

This fetches the logger named “practical”. If a logger by that name does not exist, a new logger is created; otherwise, the existing logger instance is returned. No matter how many times you get the “practical” logger in your program, the same instance is returned.

Now that you have logging, what should you log? Well, it depends on what you are doing. If the server is operating normally, you may not want to log every single step the server takes because logging consumes resources such as space for storing log entries and server processor time for writing each entry. On the other hand, if you are trying to debug, you may want to log each and every step. To deal with this, logging typically includes the notion of the *level*, or severity, of log entries. The `Level`class encapsulates the notion of the importance of messages. Each instance of `Logger`has a current level, and each message logged has an associated level; messages with levels below the instance’s current level are discarded (i.e., not logged). Each level has an associated integer *value*, so that levels are comparable and can be ordered. Seven system-recognized instances of `Level`are defined; other, user-specific, levels can be created, but there is rarely any need to do so. The built-in levels (defined as static fields of the class `Level`) are: `severe`, `warning`, `info`, `config`, `fine`, `finer`, and `finest`.

So when you log, where do the messages go? The logger sends messages to one or more `Handlers`, which

“handle” publishing the messages. By default, a logger has a single `ConsoleHandler` that prints messages to `System.err`. You can change the handler or add additional handlers to a logger (e.g., `FileHandler`). Note that like a logger, a handler has a minimum log level, so for a message to be published its level must be above *both* the logger and handlers’ threshold. Loggers and handlers are highly configurable, including their minimum level.

An important characteristic of `Logger` for our purposes is that it is *thread-safe*—that is, its methods can be called from different threads running concurrently without requiring additional synchronization among the callers. Without this feature, different messages logged by different threads might end up being interleaved in the log!

#### *Logger: Finding/Creating*

```
static Logger getLogger(String name)
static Logger getLogger(String name, String resourceBundleName)
```

The static factory methods return the named `Logger`, creating it if necessary.

Once we have the logger, we need to ... well ... log. `Logger` provides fine-grained logging facilities that differentiate between the level and even context (method call, exception, etc.) of the message.

#### *Logger: Logging a message*

```
void severe(String msg)
void warning(String msg)
void info(String msg)
void config(String msg)
void fine(String msg)
void finer(String msg)
void finest(String msg)
void entering(String sourceClass, String sourceMethod)
void entering(String sourceClass, String sourceMethod, Object param)
void entering(String sourceClass, String sourceMethod, Object[] params)
void exiting(String sourceClass, String sourceMethod)
void exiting(String sourceClass, String sourceMethod, Object result)
void throwing(String sourceClass, String sourceMethod, Throwable thrown)
void log(Level level, String msg)
void log(Level level, String msg, Throwable thrown)
```

The `severe()`, `warning()`, etc. methods log the given message at the level specified by the method name. The `entering()` and `exiting()` methods log entering and exiting the given method from the given class. Note that you may optionally specify additional information such as parameters and return values. The `throwing()` method logs an exception thrown in a specific method. The `log()` methods provide a generic logging method where level, message, and (optionally) exception can be logged. Note that many other logging methods exist; we are only noting the major types here.

We may want to customize our logger by setting the minimum logging level or the handlers for logging messages.

#### *Logger: Setting/Getting the level and handlers*

```
Handler[] getHandlers()
void addHandler(Handler handler)
void removeHandler(Handler handler)
Level getLevel()
void setLevel(Level newLevel)
boolean isLoggable(Level level)
```

The `getHandlers()` method returns an array of all handlers associated with the logger. The `addHandler()` and `removeHandler()` methods allow addition/removal of handlers to/from the logger. The `getLevel()` and `setLevel()` methods get/set the minimum logging level. The `isLoggable()` method returns true if the given level will be logged by the logger.

We are now ready to introduce some different approaches to concurrent servers.

### **3.2.7.2 Thread-per-Client**

In a *thread-per-client* server, a new thread is created to handle each connection. The server executes a loop that runs forever, listening for connections on a specified port and repeatedly accepting an incoming connection from a client and then spawning a new thread to handle that connection.

`TCPEchoServerThread.java` implements the thread-per-client architecture. It is very similar to the iterative server, using a single loop to receive and process client requests. The main difference is that it creates a thread to handle the connection instead of handling it directly. (This is possible because `EchoProtocol` implements the `Runnable` interface.) Thus, when several clients connect at approximately the same time, later ones do not have to wait for the server to finish with the earlier ones before they get service. Instead, they all appear to receive service (albeit at a somewhat slower rate) at the same time.

### **TCPEchoServerThread.java**

```
0 import java.io.IOException;
1 import java.net.ServerSocket;
2 import java.net.Socket;
3 import java.util.logging.Logger;
4
5 public class TCPEchoServerThread {
6
7 public static void main(String[] args) throws IOException {
8
9     if (args.length != 1) { // Test for correct # of args
10         throw new IllegalArgumentException("Parameter(s): <Port>");
11     }
12
13     int echoServPort = Integer.parseInt(args[0]); // Server port
14
15     // Create a server socket to accept client connection requests
16     ServerSocket servSock = new ServerSocket(echoServPort);
17
18     Logger logger = Logger.getLogger("practical");
19
20     // Run forever, accepting and spawning a thread for each connection
21     while (true) {
22         Socket clntSock = servSock.accept(); // Block waiting for connection
23         // Spawn thread to handle new connection
24         Thread thread = new Thread(new EchoProtocol(clntSock, logger));
25         thread.start();
26         logger.info("Created and started Thread " + thread.getName());
27     }
28 /* NOT REACHED */
29 }
30 }
```

1. Parameter parsing and server socket/logger creation: lines 9–18

2. Loop forever, handling incoming connections: lines 21–27

- Accept an incoming connection: line 22

- Create a new instance of Threadto handle the new connection: line 24

Since EchoProtocolimplements the Runnable interface, we can give our new instance to the Thread constructor, and the new thread will execute the run() method of EchoProtocol (which calls handleEchoClient()) when start() is invoked.

- Start the new thread for the connection and log it: lines 25–26 The getName() method of Threadreturns a String containing a name for the new thread.

### **3.2.7.3 Thread Pool**

Every new thread consumes system resources: spawning a thread takes CPU cycles and each thread has its own data structures (e.g., stacks) that consume system memory. In addition, when one thread *blocks*, the JVM saves its state, selects another thread to run, and restores the state of the chosen thread in what is called a *contextswitch*. As the number of threads increases, more and more system resources are consumed by thread overhead. Eventually, the system is spending more time dealing with context switching and thread management than with servicing connections. At that point, adding an additional thread may actually *increase* client service time.

We can avoid this problem by limiting the total number of threads and reusing threads. Instead of spawning a new thread for each connection, the server creates a *threadpool*on start-up by spawning a fixed number of threads. When a new client connection arrives at the server, it is assigned to a thread from the pool. When the thread finishes with the client, it returns to the pool, ready to handle another request. Connection requests that arrive when all threads in the pool are busy are queued to be serviced by the next available thread.

Like the thread-per-client server, a thread-pool server begins by creating a ServerSocket. Then it spawns  $N$  threads, each of which loops forever, accepting connections from the (shared) ServerSocket instance. When multiple threads simultaneously call accept() on the same ServerSocket instance, they all block until a connection is established. Then the system selects one thread, and the Socket instance for the new connection is returned *only in that thread*. The other threads remain blocked until the next connection is established and another lucky winner is chosen.

Since each thread in the pool loops forever, processing connections one by one, a thread-pool server is really like

a set of iterative servers. Unlike the thread-per-client server, a thread-pool thread does not terminate when it finishes with a client. Instead, it starts over again, blocking on accept(). An example of the thread-pool paradigm is shown in TCPEchoServerPool.java.

### **TCPEchoServerPool.java**

```

0 import java.io.IOException;
1 import java.net.ServerSocket;
2 import java.net.Socket;
3 import java.util.logging.Level;
4 import java.util.logging.Logger;
5
6 public class TCPEchoServerPool {
7
8     public static void main(String[] args) throws IOException {
9
10    if (args.length != 2) { // Test for correct # of args
11        throw new IllegalArgumentException("Parameter(s): <Port> <Threads>");
12    }
13
14    int echoServPort = Integer.parseInt(args[0]); // Server port
15    int threadPoolSize = Integer.parseInt(args[1]);
16
17    // Create a server socket to accept client connection requests
18    final ServerSocket servSock = new ServerSocket(echoServPort);
19
20    final Logger logger = Logger.getLogger("practical");
21
22    // Spawn a fixed number of threads to service clients
23    for (int i = 0; i < threadPoolSize; i++) {
24        Thread thread = new Thread() {
25            public void run() {
26                while (true) {
27                    try {
28                        Socket clntSock = servSock.accept(); // Wait for a connection
29                        EchoProtocol.handleEchoClient(clntSock, logger); // Handle it
30                    } catch (IOException ex) {
31                        logger.log(Level.WARNING, "Client accept failed", ex);
32                    }
33                }
34            }
35        };
36        thread.start();
37        logger.info("Created and started Thread = " + thread.getName());
38    }
39 }
40 }
```

1. Setup: lines 10–20 The port number to listen on and the number of threads are both passed as arguments to main(). After parsing them we create the ServerSocket and Logger instances. Note that both have to be declared final, because they are referenced inside the anonymous class instance created below.

2. Create and start *threadPoolSize* new threads: lines 23–38 For each loop iteration, an instance of an anonymous class that extends Thread is created. When the start() method of this instance is called, the thread executes the run() method of this anonymous class. The run() method loops forever, accepting a connection and then giving it to EchoProtocol for service.

- Accept an incoming connection: line 28 Since there are  $N$  different threads executing the same loop, up to  $N$  threads can be blocked on servSock's accept(), waiting for an incoming connection. The system ensures that only one thread gets a Socket for any particular connection. If no threads are blocked on accept() when a client connection is established (that is, if they are all busy servicing other connections), the new connection is queued by the system until the next call to accept() (see Section 6.4.1).

- Pass the client socket to EchoProtocol.handleEchoClient: line 29 The handleEchoClient() method encapsulates knowledge of the protocol details. It logs the connection when it finishes, as well as any exceptions encountered along the way.

- Handle exception from accept(): line 31

Since threads are reused, the thread-pool solution only pays the overhead of thread creation  $N$  times, irrespective of the total number of client connections. Since we control the maximum number of simultaneously executing threads, we can control scheduling and resource overhead. Of course, if we spawn too few threads, we can still have clients waiting a long time for service; therefore, the size of the thread pool needs to be tuned to the load, so that client connection time is minimized. The ideal would be a dispatching facility that expands the thread pool

(up to a limit) when the load increases, and shrinks it to minimize overhead during times when the load is light. It turns out that Java has just such a facility; we describe it in the next section.

### 3.2.7.4 System-Managed Dispatching: The Executor Interface

In the previous subsections, we have seen that encapsulating the details of the client-server protocol (as in EchoProtocol.java) lets us use different “dispatching” methods with the same protocol implementation (e.g., TCPEchoServerThread.java and TCPEchoServerThreadPool.java). In fact the same thing is true for the dispatching methods themselves. The interface Executor (part of the java.util.concurrent package) represents an object that executes Runnable instances according to some strategy, which may include details about queueing and scheduling, or how jobs are selected for execution. The Executor interface specifies a single method:

```
interface Executor {  
    void execute(Runnable task);  
}
```

Java provides a number of built-in implementations of Executor that are convenient and simple to use, and others that are extensively configurable. Some of these offer handling for messy details like thread maintenance. For example, if a thread stops because of an uncaught exception or other failure, they automatically spawn a new thread to replace it.

The ExecutorService interface extends Executor to provide a more sophisticated facility that allows a service to be shut down, either gracefully or abruptly. ExecutorService also allows for tasks to return a result, through the Callable interface, which is like Runnable, only with a return value.

Instances of ExecutorService can be obtained by calling various static factory methods of the convenience class Executors. The program TCPEchoServerExecutor.java illustrates the use of the basic Executor facilities.

#### **TCPEchoServerExecutor.java**

```
0 import java.io.IOException;  
1 import java.net.ServerSocket;  
2 import java.net.Socket;  
3 import java.util.concurrent.Executor;  
4 import java.util.concurrent.Executors;  
5 import java.util.logging.Logger;  
6  
7 public class TCPEchoServerExecutor {  
8  
9     public static void main(String[] args) throws IOException {  
10         if (args.length != 1) { // Test for correct # of args  
11             throw new IllegalArgumentException("Parameter(s): <Port>");  
12         }  
13  
14         int echoServPort = Integer.parseInt(args[0]); // Server port  
15  
16         // Create a server socket to accept client connection requests  
17         ServerSocket servSock = new ServerSocket(echoServPort);  
18  
19         Logger logger = Logger.getLogger("practical");  
20  
21         Executor service = Executors.newCachedThreadPool(); // Dispatch svc  
22  
23         // Run forever, accepting and spawning a thread for each connection  
24         while (true) {  
25             Socket clntSock = servSock.accept(); // Block waiting for connection  
26             service.execute(new EchoProtocol(clntSock, logger));  
27         }  
28         /* NOT REACHED */  
29     }  
30 }  
31 }
```

1 Setup: lines 11–20 The port is the only argument. We create the ServerSocket and Logger instances as before; they need not be declared final here, because we do not need an anonymous Thread subclass.

2 Get an Executor: line 22 The static factory method newCachedThreadPool() of class Executors creates an instance of ExecutorService. When its execute() method is invoked with a Runnable instance, the executor service creates a new thread to handle the

task if necessary. However, it first tries to reuse an existing thread. When a thread has been idle for at least 60 seconds, it is removed from the pool. This is almost always going to be more efficient than either of the last two TCPEchoServer\* examples.

3 Loop forever, accepting connections and executing them: lines 25–28 When a new connection arrives, a new EchoProtocolinstance is created and passed to the execute() method of *service*, which either hands it off to an already-existing thread or creates a new thread to handle it. Note that in the steady state, the cached thread pool Executorservice ends up having about the right number of threads, so that each thread stays busy and creation/destruction of threads is rare.

Once we have a server designed to use Executorfor dispatching clients, we can change dispatching strategies simply by changing the kind of Executorwe instantiate. For example, if we wanted to use a fixed-size thread pool as in our TCPEchoServerPool.javaexample, it is a matter of changing one line associated with setting the dispatch service:

```
Executor service = Executors.newFixedThreadPool(threadPoolsSize);
```

We could switch to a single thread to execute all connections either by specifying a pool size of 1, or by the following call:

```
Executor service = Executors.newSingleThreadExecutor();
```

In the Executor approach, if the single “worker” thread dies because of some failure, the Executor wil replace it with a new thread. Also, tasks are queued inside the Executor, instead of being queued inside the networking system, as they were in our original server. Note that we’ve only scratched the surface of Java’s concurrency package.

### 3.2.7.5 Blocking and Timeouts

Socket I/O calls may block for several reasons. Data input methods read() and receive() block if data is not available. A write() on a TCP socket may block if there is not sufficient space to buffer the transmitted data. The accept() method of ServerSocket()and the Socketconstructor both block until a connection has been established. Meanwhile, long round-trip times, high error rate connections, and slow (or deceased) servers may cause connection establishment to take a long time. In all of these cases, the method returns only after the request has been satisfied. Of course, a blocked method call halts progress of the application (and makes the thread that is running it useless).

What about a program that has other tasks to perform while waiting for call completion (e.g., updating the “busy” cursor or responding to user requests)? These programs may have no time to wait on a blocked method call. What about lost UDP datagrams? If we block waiting to receive a datagram and it is lost, we could block indefinitely. Here we explore the various blocking methods and approaches for limiting blocking behavior. In Chapter 5 we’ll encounter the more powerful nonblocking facilities available through the NIO package.

#### ***accept(), read(), and receive()***

For these methods, we can set a bound on the maximum time (in milliseconds) to block, using the setSoTimeout() method of Socket, ServerSocket, and DatagramSocket. If the specified time elapses before the method returns, an InterruptedIOException is thrown. For Socketinstances, we can also use the available() method of the socket’s InputStreamto check for available data before calling read().

#### ***Connecting and Writing***

The Socket constructor attempts to establish a connection to the host and port supplied as arguments, blocking until either the connection is established or a system-imposed timeout occurs. Unfortunately, the system-imposed timeout is long, and Java does not provide any means of shortening it. To fix this, call the parameterless constructor for Socket, which returns an unconnected instance. To establish a connection, call the connect() method on the newly constructed socket and specify both a remote endpoint and timeout (milliseconds).

A write() call blocks until the last byte written is copied into the TCP implementation’s local buffer; if the available buffer space is smaller than the size of the write, some data must be successfully transferred to the other end of the connection before the call to write() will return. Thus, the amount of time that a write()may block is ultimately controlled by the receiving application. Unfortunately, Java currently does not provide any way to cause a write() to time out, nor can it be interrupted by another thread. Therefore, any protocol that sends a large enough amount of data over a Socketinstance can block for an unbounded amount of time.

#### ***Limiting Per-Client Time***

Suppose we want to implement the Echo protocol with a limit on the amount of time taken to service each client. That is, we define a target, timelimit, and implement the protocol in such a way that after time limit milliseconds, the protocol instance is terminated.The protocol instance keeps track of the amount of time remaining, and uses setSoTimeout() to ensure that no read() all blocks for longer than that time. Since there is no way to bound the

duration of a write() call, we cannot really guarantee that the time limit will hold. Nevertheless, TimeLimitEchoProtocol.java implements this approach; to use it with TCPEchoServerExecutor.java, simply change the second line of the body of the while loop to:

```
service.execute(new TimeLimitEchoProtocol(clntSock, logger));
```

More powerful mechanisms can limit the time that threads can block—on all I/O calls, including writes—using the facilities of the NIO package.

### TimeLimitEchoProtocol.java

```
0 import java.io.IOException;
1 import java.io.InputStream;
2 import java.io.OutputStream;
3 import java.net.Socket;
4 import java.util.logging.Level;
5 import java.util.logging.Logger;
6
7 class TimelimitEchoProtocol implements Runnable {
8     private static final int BUFSIZE = 32; // Size (bytes) of buffer
9     private static final String TIMELIMIT = "10000"; // Default limit (ms)
10    private static final String TIMELIMITPROP = "Timelimit"; // Property
11
12    private static int timelimit;
13    private Socket clntSock;
14    private Logger logger;
15
16    public TimelimitEchoProtocol(Socket clntSock, Logger logger) {
17        this.clntSock = clntSock;
18        this.logger = logger;
19        // Get the time limit from the System properties or take the default
20        timelimit = Integer.parseInt(System.getProperty(TIMELIMITPROP, TIMELIMIT));
21    }
22
23    public static void handleEchoClient(Socket clntSock, Logger logger) {
24
25        try {
26            // Get the input and output I/O streams from socket
27            InputStream in = clntSock.getInputStream();
28            OutputStream out = clntSock.getOutputStream();
29            int recvMsgSize; // Size of received message
30            int totalBytesEchoed = 0; // Bytes received from client
31            byte[] echoBuffer = new byte[BUFSIZE]; // Receive buffer
32            long endTime = System.currentTimeMillis() + timelimit;
33            int timeBoundMillis = timelimit;
34
35            clntSock.setSoTimeout(timeBoundMillis);
36            // Receive until client closes connection, indicated by -1
37            while ((timeBoundMillis > 0) && // catch zero values
38                   ((recvMsgSize = in.read(echoBuffer)) != -1)) {
39                out.write(echoBuffer, 0, recvMsgSize);
40                totalBytesEchoed += recvMsgSize;
41                timeBoundMillis = (int) (endTime - System.currentTimeMillis());
42                clntSock.setSoTimeout(timeBoundMillis);
43            }
44            logger.info("Client " + clntSock.getRemoteSocketAddress() +
45                       ", echoed " + totalBytesEchoed + " bytes.");
46        } catch (IOException ex) {
47            logger.log(Level.WARNING, "Exception in echo protocol", ex);
48        }
49    }
50
51    public void run() {
52        handleEchoClient(this.clntSock, this.logger);
53    }
54 }
```

The TimelimitEchoProtocol class is similar to the EchoProtocol class, except that it attempts to bound the total time an echo connection can exist to 10 seconds. At the time the handleEchoClient() method is invoked, a deadline is computed using the current time and the time bound. After each read(), the time between the current time and the deadline is computed, and the socket timeout is set to the remaining time.

### 3.2.7.6 Multiple Recipients

So far all of our sockets have dealt with communication between exactly two entities, usually a server and a client. Such one-to-one communication is sometimes called *unicast*. Some information is of interest to multiple recipients. In such cases, we could unicast a copy of the data to each recipient, but this may be very inefficient. Unicasting multiple copies over a single network connection wastes bandwidth by sending the same information multiple times. In fact, if we want to send data at a fixed rate, the bandwidth of our network connection defines a hard limit on the number of receivers we can support. For example, if our video server sends 1Mbps streams and its network connection is only 3Mbps (a healthy connection rate), we can only support three simultaneous users.

Fortunately, networks provide a way to use bandwidth more efficiently. Instead of making the sender responsible for duplicating packets, we can give this job to the network. In our video server example, we send a single copy of the stream across the server's connection to the network, which then duplicates the data only when appropriate. With this model of duplication, the server uses only 1Mbps across its connection to the network, irrespective of the number of clients.

There are two types of one-to-many service: *broadcast* and *multicast*. With broadcast, all hosts on the (local) network receive a copy of the message. With multicast, the message is sent to a *multicast address*, and the network delivers it only to those hosts that have indicated that they want to receive messages sent to that address. In general, only UDP sockets are allowed to broadcast or multicast.

#### Broadcast

Broadcasting UDP datagrams is similar to unicasting datagrams, except that a *broadcast address* is used instead of a regular (unicast) IP address. Note that IPv6 does not explicitly provide broadcast addresses; however, there is a special all-nodes, link-local-scope multicast address, FFO2::1, that multicasts to all nodes on a link. The IPv4 *local broadcast* address (255.255.255.255) sends the message to every host on the same broadcast network. Local broadcast messages are never forwarded by routers. A host on an Ethernet network can send a message to all other hosts on that same Ethernet, but the message will not be forwarded by a router. IPv4 also specifies *directed broadcast* addresses, which allow broadcasts to all hosts on a specified network; however, since most Internet routers do not forward directed broadcasts, we do not deal with them here.

There is no network wide broadcast address that can be used to send a message to all hosts. To see why, consider the impact of a broadcast to every host on the Internet. Sending a single datagram would result in a very, very large number of packet duplications by the routers, and bandwidth would be consumed on each and every network. The consequences of misuse (malicious or accidental) are too great, so the designers of IP left such an Internet wide broadcast facility out on purpose.

Even so, local broadcast can be very useful. Often, it is used in state exchange for network games where the players are all on the same local (broadcast) network. In Java, the code for unicasting and broadcasting is the same. To play with broadcasting applications, we can simply use our VoteClientUDP.java with a broadcast destination address.

#### Multicast

As with broadcast, one of the main differences between multicast and unicast is the form of the address. A multicast address identifies a set of receivers. The designers of IP allocated a range of the address space dedicated to multicast, specifically 224.0.0.0 to 239.255.255 for IPv4 and any address starting with FF for IPv6. With the exception of a few reserved multi-cast addresses, a sender can send datagrams addressed to any address in this range. In Java, multicast applications generally communicate using an instance of MulticastSocket, a subclass of DatagramSocket. It is important to understand that a MulticastSocket is actually a UDP socket (DatagramSocket), with some extra multicast-specific attributes that can be controlled. Our next examples implement a multicast sender and receiver of vote messages.

#### VoteMulticastSender.java

```
0 import java.io.IOException;
1 import java.net.DatagramPacket;
2 import java.net.InetAddress;
3 import java.net.MulticastSocket;
4
```

```

5 public class VoteMulticastSender {
6
7     public static final int CANDIDATEID = 475;
8
9     public static void main(String args[]) throws IOException {
10
11         if ((args.length < 2) || (args.length > 3)) { // Test # of args
12             throw new IllegalArgumentException("Parameter(s):<Multicast Addr><Port>[<TTL>]");
13         }
14
15         InetAddress destAddr = InetAddress.getByName(args[0]); // Destination
16         if (!destAddr.isMulticastAddress()) { // Test if multicast address
17             throw new IllegalArgumentException("Not a multicast address");
18         }
19
20         int destPort = Integer.parseInt(args[1]); // Destination port
21         int TTL = (args.length == 3) ? Integer.parseInt(args[2]) : 1; // Set TTL
22
23         MulticastSocket sock = new MulticastSocket();
24         sock.setTimeToLive(TTL); // Set TTL for all datagrams
25
26         VoteMsgCoder coder = new VoteMsgTextCoder();
27
28         VoteMsg vote = new VoteMsg(true, true, CANDIDATEID, 1000001L);
29
30         // Create and send a datagram
31         byte[] msg = coder.toWire(vote);
32         DatagramPacket message=new DatagramPacket(msg,msg.length,destAddr,destPort);
33         System.out.println("Sending Text-Encoded Request ("+msg.length+" bytes): ");
34         System.out.println(vote);
35         sock.send(message);
36
37         sock.close();
38     }
39 }
```

The only significant differences between our unicast and multicast senders are that 1) we verify that the given address is multicast, and 2) we set the initial Time To Live (TTL) value for the multicast datagram. Every IP datagram contains a TTL, initialized to some default value and decremented (usually by one) by each router that forwards the packet. When the TTL reaches zero, the packet is discarded. By setting the initial value of the TTL, we limit the distance a packet can travel from the sender. The rules for multicast TTL are actually not quite so simple. It is not necessarily the case that a packet with TTL = 4 can travel four hops from the sender; however, it will not travel *more* than four hops.

Unlike broadcast, network multicast duplicates the message only to a specific set of receivers. This set of receivers, called a *multicastgroup*, is identified by a shared multicast (or group) address. Receivers need some mechanism to notify the network of their interest in receiving data sent to a particular multicast address, so that the network can forward packets to them. This notification, called *joining a group*, is accomplished with the joinGroup() method of MulticastSocket. Our multicast receiver joins a specified group, receives and prints a single multicast message from that group, and exits.

### **VoteMulticastReceiver.java**

```

0 import java.io.IOException;
1 import java.net.DatagramPacket;
2 import java.net.InetAddress;
3 import java.net.MulticastSocket;
4 import java.util.Arrays;
5
6 public class VoteMulticastReceiver {
7
8     public static void main(String[] args) throws IOException {
9
10        if (args.length != 2) { // Test for correct # of args
11            throw new IllegalArgumentException("Parameter(s): <Multicast Addr> <Port>");
12        }
13    }
```

```

14     InetAddress address = InetAddress.getByName(args[0]); // Multicast address
15     if (!address.isMulticastAddress()) { // Test if multicast address
16         throw new IllegalArgumentException("Not a multicast address");
17     }
18
19     int port = Integer.parseInt(args[1]); // Multicast port
20     MulticastSocket sock = new MulticastSocket(port); // for receiving
21     sock.joinGroup(address); // Join the multicast group
22
23     VoteMsgTextCoder coder = new VoteMsgTextCoder();
24
25     // Receive a datagram
26     DatagramPacket packet=newDatagramPacket(
27         new byte[VoteMsgTextCoder.MAX_WIRE_LENGTH],VoteMsgTextCoder.MAX_WIRE_LENGTH);
28     sock.receive(packet);
29
30     VoteMsg vote = coder.fromWire( Arrays.copyOfRange(packet.getData(), 0, packet
31             .getLength()));
32
33     System.out.println("Received Text-Encoded Request (" + packet.getLength()
34             + " bytes): ");
35     System.out.println(vote);
36
37     sock.close();
38 }
39 }
```

The only significant difference between our multicast and unicast receiver is that the multicast receiver must join the multicast group by supplying the desired multicast address.

Multicast datagrams can, in fact, be sent from a DatagramSocket by simply using a multicast address. However, a MulticastSocket has a few capabilities that a DatagramSocket does not, including 1) allowing specification of the datagram TTL, and 2) allowing the interface through which datagrams are sent to the group to be specified/changed (an interface is identified by its Internet address). A multicast receiver, on the other hand, *must* use a MulticastSocket because it needs the ability to join a group.

MulticastSocket is a subclass of DatagramSocket, so it provides all of the DatagramSocket methods. We only present methods specific to or adapted for MulticastSocket.

#### *MulticastSocket: Creation*

```

MulticastSocket()
MulticastSocket(int localPort)
MulticastSocket(SocketAddress bindaddr)
```

These constructors create a multicast-capable UDP socket. If the local port is not specified, or is specified as 0, the socket is bound to any available local port. If the address is specified, the socket is restricted to receiving only on that address.

If we wish to receive any datagrams, we need to join a multicast group.

#### *MulticastSocket: Group management*

```

void joinGroup(InetAddress groupAddress)
void joinGroup(SocketAddress mcastaddr, NetworkInterface netIf)
void leaveGroup(InetAddress groupAddress)
void leaveGroup(SocketAddress mcastaddr, NetworkInterface netIf)
```

The joinGroup() and leaveGroup() methods manage multicast group membership. A socket may be a member of multiple groups simultaneously. Joining a group of which this socket is already a member or leaving a group of which this socket is not a member may generate an exception. Optionally, you may specify an interface on which to join/ leave.

#### *MulticastSocket: Setting/Getting multicast options*

```

int getTimeToLive()
void setTimeToLive(int ttl)
boolean getLoopbackMode()
void setLoopbackMode(boolean disable)
InetAddress getInterface()
NetworkInterface getNetworkInterface()
void setInterface(InetAddress inf)
void setNetworkInterface(NetworkInterface netIf)
```

The getTimeToLive() and setTimeToLive() methods get and set the time-to-live for all datagrams sent on this socket. A socket with

loopback mode enabled will receive the data-grams it sends. The `getLoopbackMode()` and `setLoopbackMode()` methods set the loopback mode for the multicast socket where setting the loopback mode to true disables loopback. The `getInterface()`, `setInterface()`, `getNetworkInterface()`, `setNetworkInterface()` methods set the outgoing interface used in sending multicast packets. This is primarily used on hosts with multiple interfaces. The default multicast interface is platform dependent.

The decision to use broadcast or multicast depends on several factors, including the network location of receivers and the knowledge of the communicating parties. The scope of a broadcast on the Internet is restricted to a local broadcast network, placing severe restrictions on the location of the broadcast receivers. Multicast communication may include receivers anywhere in the network, so multicast has the advantage that it can cover a distributed set of receivers. At the time of writing of this book, there are severe limitations on who can receive multicast traffic on the Internet. Multicast should work if the sender and receivers are on the same LAN. The disadvantage of IP multicast is that receivers must know the address of a multicast group to join. Knowledge of an address is not required to receive broadcast. In some contexts, this makes broadcast a better mechanism than multicast for discovery. All hosts can receive broadcast by default, so it is simple to ask all hosts on a single network a question like “Where’s the printer?”

UDP unicast, multicast, and broadcast are all implemented using an underlying UDP socket. The semantics of most implementations are such that a UDP datagram will be delivered to all sockets bound to the destination port of the packet. That is, a `DatagramSocket` or `Multi-castSocket` instance bound to a local port X (with local address not specified, i.e., a wild card), on a host with address Y will receive any UDP datagram destined for port X that is 1) unicast with destination address Y, 2) multicast to a group that *any* application on Y has joined, or 3) broadcast where it can reach host Y. A receiver can use `connect()` to limit the datagram source address and port. Also, a `DatagramSocket` can specify the local unicast address, which prevents delivery of multicast and broadcast packets. See `UDPEchoClientTimeout.java` for an example of destination address verification.

### 3.2.7.7 Exercises

- 1 State precisely the conditions under which an iterative server is preferable to a multiprocessing server.
- 2 Would you ever need to implement a timeout in a client or server that uses TCP?
- 3 How can you determine the minimum and maximum allowable sizes for a socket’s send and receive buffers? Determine the minimums for your system.

### **3.2.8 Distributing Objects**

Distributed objects are a potentially powerful tool that has only become broadly available for developers at large in the past few years. The power of distributing objects is not in the fact that a bunch of objects are scattered across the network. The power lies in that any agent in your system can directly interact with an object that "lives" on a remote host. Distributed objects, if they're done right, really give you a tool for opening up your distributed system's resources across the board. And with a good distributed object scheme you can do this as precisely or as broadly as you'd like.

The first three sections of this chapter go over the motivations for distributing objects, what makes distributed object systems so useful, and what makes up a "good" distributed object system. Readers who are already familiar with basic distributed object issues can skip these sections and go on to the following sections, where we discuss two major distributed object protocols that are available in the Java environment: CORBA and RMI.

Although this chapter will cover the use of both RMI and CORBA for distributing objects, the rest of the book primarily uses examples that are based on RMI, where distributed objects are needed. We chose to do this because RMI is a simpler API and lets us write relatively simple examples that still demonstrate useful concepts, without getting bogged down in CORBA API specifics. Some of the examples, if converted to be used in production environments, might be better off implemented in CORBA.

#### **3.2.8.1 Why Distribute Objects?**

We discussed some of the optimal data/function partitioning capabilities that you'd like to have available when developing distributed applications. These included being able to distribute data/function "modules" freely and transparently, and have these modules be defined based on application structure rather than network distribution influences. Distributed object systems try to address these issues by letting developers take their programming objects and have them "run" on a remote host rather than the local host. The goal of most distributed object systems is to let any object reside anywhere on the network, and allow an application to interact with these objects exactly the same way as they do with a local object. Additional features found in some distributed object schemes are the ability to construct an object on one host and transmit it to another host, and the ability for an agent on one host to create a new object on another host.

The value of distributed objects is more obvious in larger, more complicated applications than in smaller, simpler ones. That's because much of the trade-off between distributed objects and other techniques, like message passing, is between simplicity and robustness. In a smaller application with just a few object types and critical operations, it's not difficult to put together a catalog of simple messages that would let remote agents perform all of their critical operation through on-line transactions. With a larger application, this catalog of messages gets complicated and difficult to maintain. It's also more difficult to extend a large message-passing system if new objects and operations are added. So being able to distribute the objects in our system directly saves us a lot of design overhead, and makes a large distributed system easier to maintain in the long run.

#### **3.2.8.2 Creating Remote Objects**

The essential requirements in a distributed object system are the ability to create or invoke objects on a remote host or process, and interact with them as if they were objects within our own process. It seems logical that we would need some kind of message protocol for sending requests to remote agents to create new objects, to invoke methods on these objects, and to delete the objects when we're done with them. The networking support in the Java API makes it very easy to implement a message protocol. But what kinds of things does a message protocol have to do if it's supporting a distributed object system?

To create a remote object, we need to reference a class, provide constructor arguments for the class, and receive a reference to the created object in return. This object reference will be used to invoke methods on the object, and eventually to ask the remote agent to destroy the object when we are done with it. So the data we will need to send over the network include *class references*, *object references*, *method references*, and *method arguments*.

The `ClassLoader` can be used to send class definitions over the network. If we want to create a new remote object from a given class, we can send the class definition to the remote host, and tell it to build one using a default constructor. Object references require some thought, though. These are not the same as local Java object references. We need to have an object reference that we can package up and send over the network, i.e., one that's *Serializable*. This object reference, once we receive it, will still need to refer back to the original object on the

remote host, so that when we call methods on it the method invocations are deferred to the "source" object on the remote host. One simple way to implement remote object references is to build an object lookup table into the remote agent. When a client requests a new object, the remote agent builds the requested object, puts the object into the table, and sends the table index of the object to the client. If we use sockets and streams to send requests and object references back and forth between remote agents, a client might request a remote object with something like this:

```
Class myClass = Class.forName("Myclass");
Socket objConn = new Socket("object.server.net", 1234);
OutputStreamWriter out = new OutputStreamWriter(objConn.getOutputStream());
DataInputStream in = new DataInputStream(objConn.getInputStream());
out.write("new " + myClass.getName());
int objRef = in.readInt();
```

The integer `objRef` returned by the remote server can be used to reference the new remote object. On the other end of the socket, the agent receiving the request for the remote object may handle the request like this:

```
Hashtable objTable = new Hashtable();
ServerSocket server = ...;
Socket conn;
// Accept the connection from the client
if ((conn = server.accept()) != null) {
    DataOutputStream out =
        new DataOutputStream(conn.getOutputStream());
    BufferedReader in = new BufferedReader(
        new InputStreamReader(conn.getInputStream()));
    String cmd = in.readLine();
    // Parse the command type from the command string
    if (parseCmd(cmd).compareTo("new") == 0) {
        // The client wants a new object created,
        // so parse the class name from the command string
        String classname = parseClass(cmd);
        // Create the Class object and make an instance
        Class reqClass = Class.forName(classname);
        Object obj = reqClass.newInstance();
        // Register the object and return the integer
        // identifier/reference to the client
        Integer objID = nextID();
        objTable.put(objID, obj);
        out.writeInt(objID.intValue());
    }
}
```

The object server reads the class name sent by the requestor, looks up the class using the static `Class.forName()` method, and creates a new instance of the class by calling the `newInstance()` method on the `Class` object. Once the object has been created, the server generates a unique identifier for the object and sends it back to the requestor. Note that we've already limited our remote object scheme, by forcing the use of default constructors, e.g., those with no arguments. The remote host creates the requested object by calling `newInstance()` on its class, which is equivalent to creating the object by calling the class constructor with no arguments. Since we don't (yet) have a way to specify methods on classes over the network, or a way to send arguments to these methods, we have to live with this limitation for now.

### 3.2.8.3 Remote Method Calls

Now that the requestor has a reference to an object on the remote host, it needs a way to invoke methods on the object. Since Java allows us to query a class or object for its declared methods and data members, the local agent can get a direct reference to the method that it wants to invoke on the remote object, in the form of a `Method` object:

```
Class reqClass = Class.forName("Myclass");
Method reqMethod = reqClass.getDeclaredMethod("getName", null);
```

In this example, the local agent has retrieved a reference to a getName() method, with no arguments, on the class Myclass. It could now use this method reference to call the method on a local Myclass instance:

```
Myclass obj = new Myclass();
reqMethod.invoke(obj, null);
```

This may seem like a roundabout way to accomplish the same thing as calling obj.getName() on the Myclass object, and it is. But in order to call a method on our remote object, we need to send a reference to the method over the network to the remote host. One way to do this is to assign identifiers to all of the methods on the class, just like we did for remote objects. Since both the object requestor and the object server can get a list of the class's methods by calling the getDeclaredMethods() method on the class, we could simply use the index of the method in the returned list as its identifier. Then the object requestor can call a method on a remote object by simply sending the remote host the object's identifier, and the identifier for the method to call. Assuming that our local agent has the same object reference from the earlier example, the remote method call would look something like this:

```
Method reqMethod = reqClass.getDeclaredMethod("getName", null);
Method[] methodList = reqClass.getDeclaredMethods();
int methodIdx = 0;
for (int i = 0; i < methodList.length; i++) {
    if (reqMethod == methodList[i]) {
        methodIdx = i;
        break;
    }
}
String cmd = "call " + methodIdx + " on " + objRef;
out.writeUTF(cmd);
```

This approach to handling remote method invocation is a general one; it will work for any class that we want to distribute. So far so good. But what about the arguments to the remote methods? And what about the return value from the remote method? Our example used a getName() method with no arguments, but if the method does take arguments, we'll need to send these to the remote host as well. We can also assume that a method called "getName" will probably return some kind of Stringvalue, which we'll need to get back from the remote host. This same problem exists in the creation of the remote object. With our method reference scheme we can now specify which constructor to use when the remote host creates our object, but we still need a way to send the constructor arguments to the remote host.

By now this exercise is beginning to look a lot more serious than we might have expected. In distributed object systems, the task of packaging up method arguments for delivery to the remote object, and the task of gathering up method return values for the client, are referred to as *data marshaling*. One approach we can take to data marshaling is to turn every object argument in a remote method call into a remote object just like we did previously, by generating an object reference and sending that to the remote agent as the method argument. If the method returns an object value, then the remote host can generate a new object reference and send that back to the local host. So now the remote host and the local host are acting as both object servers and object requestors. We started out with the remote host creating objects for the local host to invoke methods on, but now the local host is "serving" objects for method arguments, and the remote host is serving a bunch of new objects for method return values. And if the remote host needs to call any methods on objects that are arguments to other methods, or if the local host needs to call methods on object return values, then we'll need to send method references back and forth for these remote method calls as well.

To further complicate matters, we also have to worry about situations where you don't want a remote object reference sent as the method argument. In some cases, you may want to send objects by copy rather than by reference. In other words, you may just need to send the value of the object from one host to another, and not want changes to an object propagated back to the original source object. How do we serialize and transmit an object's value to a remote agent? One way is to tell the other agent to create a new object of the right type, as we did to create our original remote object, and then indicate the new object as the method argument or return value.

### 3.2.8.4 Other Issues

Our hypothetical remote object scheme, using object tables, integer object references based on table location, and integer method references based on the method's index/position in the class definition, is a bit ad-hoc and not very

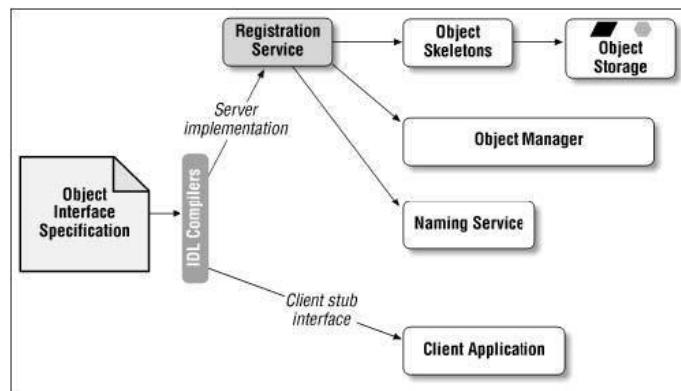
elegant. It will work, but probably not very well. For one thing, it is not very scalable in terms of development complexity and runtime complexity. Each agent on the network is maintaining its own object table and its own set of object identifiers. Each remote method call could potentially generate more entries in the object tables on both ends of the call, for method arguments and for method return values. And since there's no guarantee that two agents won't use the same identifier for two different objects, each agent using remote objects will need to keep its own table of remote object identifiers and the agent they came from. So now each agent has to maintain two object reference tables: one for objects that it is serving to other agents, and another for objects that it is using remotely. A more elegant way to handle this would be to create a naming service for objects, where an agent serving an object could register its objects with the naming service and generate a unique name/address for the object. The naming service would be responsible for mapping named objects to where they actually live. Users of the object could then find the object with one name, rather than a combination of an object ID and the object's host.

Another issue with this remote object scheme is the distribution of workload across the distributed system. In returning an object by value as the result of a method call, for example, the object server instructs the client to create the returned object value. The creation of this object could be a significant effort, depending on the type of object. Under normal, non-distributed conditions the creation of the return value is considered a part of the overhead of calling the method. You would hope that when you invoke this method remotely, all of the overhead, including the creation of the return value, would be off-loaded to the remote host. Instead, we're pushing some of the work to the remote host and keeping some locally. The same issue comes up when an agent invokes a remote method and passes method arguments by value instead of by reference. The calling agent tells the serving agent to create the method argument values on its side, which increases the net overhead on the server side for the remote method call.

Hopefully this extended thought experiment has highlighted some of the serious issues that arise when trying to distribute objects over the network. In the next section, we'll look at the features that a distributed object system needs to have in order to address these issues.

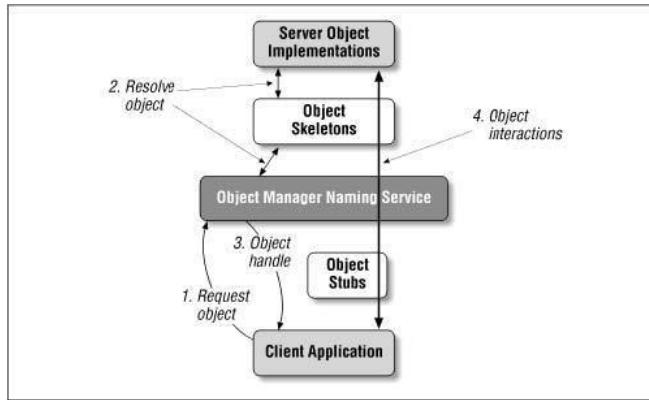
### 3.2.8.5 Features of Distributed Object Systems

From our exercise in the previous section, we uncovered some of the features that distributed object systems need. These features, plus some others, are illustrated in Figure 3.11. An object interface specification is used to generate a server *implementation* of a class of objects, an interface between the object implementation and the object manager, sometimes called an object *skeleton*, and a client interface for the class of objects, sometimes called an object *stub*. The skeleton will be used by the server to create new instances of the class of objects and to route remote method calls to the object implementation. The stub will be used by the client to route transactions (method invocations, mostly) to the object on the server. On the server side, the class implementation is passed through a *registration service*, which registers the new class with a *naming service* and an *object manager*, and then stores the class in the server's storage for object skeletons.



**Figure 3.11. General architecture for distributed object systems**

With an object fully registered with a server, the client can now request an instance of the class through the naming service. The runtime transactions involved in requesting and using a remote object are shown in Figure 3.12. The naming service routes the client's request to the server's object manager, which creates and initializes the new object using the stored object skeleton. The new object is stored in the server's object storage area, and an object handle is issued back to the client in the form of an object stub interface. This stub is used by the client to interact with the remote object.



**Figure 3.12. Remote object transactions at runtime**

While Figure 3.12 illustrates a client–server remote object environment, a remote object scheme can typically be used in a peer–to–peer manner as well. Any agent in the system can act as both a server and a client of remote objects, with each maintaining its own object manager, object skeleton storage, and object instance storage. In some systems the naming service can be shared between distributed agents, while in others each agent maintains its own naming service.

**Object Interface Specification.** To provide a truly open system for distributing objects, the distributed object system should allow the client to access objects regardless of their implementation details, like hardware platform and software language. It should also allow the object server to implement an object in whatever way it needs to. Although in this book we are talking about implementing systems in Java, you may have valuable services already implemented in C, C++, or Smalltalk, and these services might be expensive to reimplement in Java. In this situation you'd like the option of wrapping your existing services with object interfaces and using them directly via the remote object system.

Some distributed object systems provide a platform–independent means for specifying object interfaces. These object interface descriptions can be converted into server skeletons, which can be compiled and implemented in whatever form the server requires. The same object interfaces can be used to generate client–side stub interfaces. If we're dealing with a Java–based distributed system, then the server skeletons and client stubs will be generated as Java class definitions, which will then be compiled into bytecodes.

In CORBA, object interfaces are described using a platform–independent language called the Interface Definition Language (IDL). Other similar languages are the Interface Specification Language (ISL) in Xerox's Inter–Language Unification (ILU) system, and the Distributed Component Object Model (DCOM, comprised of COM and an extended version of DCE–RPC) used in Microsoft's DCOM system.

The **Object Manager** is really at the heart of the distributed object system, since it manages the object skeletons and object references on an object server. The object manager plays a role similar to that of an Object Request Broker (ORB) in a CORBA system, or the registry service in RMI, both of which will be discussed in more detail shortly. When a client asks for a new object, the object manager locates the skeleton for the class of object requested, creates a new object based on the skeleton, stores the new object in object storage, and sends a reference to the object back to the client. Remote method calls made by the client are routed through the manager to the proper object on the server, and the manager also routes the results back to the client. Finally, when the client is through with the remote object, it can issue a request to the object manager to destroy the object. The manager removes the object from the server's storage and frees up any resources the object is using.

Some distributed object systems support things like dynamic object activation and deactivation, and persistent objects. The object manager typically handles these functions for the object server. In order to support dynamic object activation and deactivation, the object manager needs to have an activation and deactivation method registered for each object implementation it manages. When a client requests activation of a new instance of an interface, for example, the object manager invokes the activation method for the implementation of the interface, which should generate a new instance. A reference to the new instance is returned to the client. A similar process is used for deactivating objects. If an object is set to be persistent, then the object manager needs a method for storing the object's state when it is deactivated, and for restoring it the next time a client asks for the object.

Depending on the architecture of the distributed object system, the object manager might be located on the host serving the objects, or its functions might be distributed between the client and the server, or it might reside completely on a third host, acting as a liaison between the object client and the object server.

The **Registration/Naming Service** acts as an intermediary between the object client and the object manager. Once we have defined an interface to an object, an implementation of the interface needs to be registered with the service so that it can be addressed by clients. In order to create and use an object from a remote host, a client needs a naming service so that it can specify things like the type of object it needs, or the name of a particular object if one already exists on the server. The naming service routes these requests to the proper object server. Once the client has an object reference, the naming service might also be used to route method invocations to their targets.

If the object manager also supports dynamic object activation and persistent objects, then the naming service can also be used to support these functions. If a client asks the service to activate a new instance of a given interface, the naming service can route this request to an object server that has an implementation of that interface. And if an object manager has any persistent objects under its control, the naming service can be notified of this so that requests for the object can be routed correctly.

**Object Communication Protocol.** In order for the client to interact with the remote object, a general protocol for handling remote object requests is needed. This protocol needs to support, at a minimum, a means for transmitting and receiving object references, method references, and data in the form of objects or basic data types. Ideally we don't want the client application to need to know any details about this protocol. It should simply interact with local object interfaces, letting the object distribution scheme take care of communicating with the remote object behind the scenes. This minimizes the impact on the client application source code, and helps you to be flexible about how clients access remote services.

**Development Tools.** Of course, we'll need to develop, debug, and maintain the object interfaces, as well as the language-specific implementations of these interfaces, which make up our distributed object system. Object interface editors and project managers, language cross-compilers, and symbolic debuggers are essential tools. The fact that we are developing distributed systems imposes further requirements on these tools, since we need a reasonable method to monitor and diagnose object systems spread across the network. Load simulation and testing tools become very handy here as well, to verify that our server and the network can handle the typical request frequencies and types we expect to see at runtime.

**Security.** As we have already mentioned, any network interactions carry the potential need for security. In the case of distributed object systems, agents making requests of the object broker may need to be authenticated and authorized to access elements of the object repository, and restricted from other areas and objects. Transactions between agents and the remote objects they are invoking may need to be encrypted to prevent eavesdropping. Ideally, the object distribution scheme will include direct support for these operations. For example, the client may want to "tunnel" the object communication protocol through a secure protocol layer, with public key encryption on either end of the transmission.

### 3.2.8.6 Distributed Object Schemes for Java

While there are several distributed object schemes that can be used within the Java environment, we'll only cover two that qualify as serious options for developing your distributed applications: CORBA and RMI. Both of them have their advantages and their limitations, which we'll look at in detail in the following sections.

During this discussion, we'll be using an example involving a generic problem solver, which we'll distribute using both CORBA and RMI. We'll show in each case how instances of this class can be used remotely using these various object distribution schemes. A Java interface for the example class, called Solver, is shown in Example 3.12. The Solver acts as a generic compute engine that solves numerical problems. Problems are given to the Solver in the form of ProblemSet objects; the ProblemSet interface is shown in Example 3.13. The ProblemSet holds all of the information describing a problem to be solved by the Solver. The ProblemSet also contains fields for the solution to the problem it represents. In our highly simplified example, we're assuming that any problem is described by a single floating-point number, and the solution is also a single floating-point value.

#### Example 3.12. A Problem Solver Interface

```
package dcj.examples;
import java.io.OutputStream;
//
// Solver:
// An interface to a generic solver that operates on ProblemSets
//
```

```

public interface Solver
{
    // Solve the current problem set
    public boolean solve();
    // Solve the given problem set
    public boolean solve(ProblemSet s, int numIters);
    // Get/set the current problem set
    public ProblemSet getProblem();
    public void setProblem(ProblemSet s);
    // Get/set the current iteration setting
    public int getIterations();
    public void setIterations(int numIter);
    // Print solution results to the output stream
    public void printResults(OutputStream os);
}

```

### Example 3.13. A Problem Set Class

```

package dcj.examples;
public class ProblemSet
{
    protected double value = 0.0;
    protected double solution = 0.0;
    public double getValue() { return value; }
    public double getSolution() { return solution; }
    public void setValue(double v) { value = v; }
    public void setSolution(double s) { solution = s; }
}

```

Our Solver interface represents a pretty simple compute engine, but it has some features meant to highlight the attributes of a distributed object scheme. As we said before, the Solver accepts problems in the form of ProblemSet objects. It also has a single compute parameter, the number of iterations used in solving the problem. Most computational algorithms have parameters that can be used to alter the way a problem is solved: basic iterative methods usually have a maximum number of iterations to run, so we're using that as the only parameter on our simplified Solver.

A Solver has two solve() methods. One has no arguments and causes the Solver to solve the default problem using the default settings. The default problem for the Solver can be set using the setProblem() method, and the default iteration limit can be set using the setIterations() method. You can also get these values using the getProblem() and getIterations() methods on the interface. The other solve() method includes arguments that give the problem to be solved, and the iteration limit to use for solving the problem.

This Solver interface acts as a sort of litmus test for distributed object schemes. It includes methods that accept Object arguments (ProblemSets, specifically), it can maintain its own state (the default problem and default iteration limit), which needs to be kept consistent across method calls from multiple clients, and it includes a method that involves generic I/O.

### 3.2.9 Remote Method Invocation (RMI)

With all our method calls so far, the objects upon which such methods have been invoked have been local. However, in a distributed environment, it is often desirable to be able to invoke methods on remote objects (i.e., on objects located on other systems). RMI (*Remote Method Invocation*) provides a platform-independent means of doing just this. Under RMI, the networking details required by explicit programming of streams and sockets disappear and the fact that an object is located remotely is almost transparent to the Java programmer. Once a reference to the remote object has been obtained, the methods of that object may be invoked in exactly the same way as those of local objects. Behind the scenes, of course, RMI will be making use of byte streams to transfer data and method invocations, but all of this is handled automatically by the RMI infrastructure. RMI has been a core component of Java from the earliest release of the language, but has undergone some evolutionary changes since its original specification.

#### 3.2.9.1 The Basic RMI Process

Though the above paragraph referred to obtaining a reference to a remote object, this was really a simplification of what actually happens. The server program that has control of the remote object registers an interface with a naming service, thereby making this interface accessible by client programs. The interface contains the signatures for those methods of the object that the server wishes to make publicly available. A client program can then use the same naming service to obtain a reference to this interface in the form of what is called a stub. This stub is effectively a local surrogate (a 'stand-in' or placeholder) for the remote object. On the remote system, there will be another surrogate called a skeleton. When the client program invokes a method of the remote object, it appears to the client as though the method is being invoked directly on the object. What is actually happening, however, is that an equivalent method is being called in the stub. The stub then forwards the call and any parameters to the skeleton on the remote machine. Only primitive types and those reference types that implement the *Serializable* interface may be used as parameters. (The serializing of these parameters is called marshalling.)

Upon receipt of the byte stream, the skeleton converts this stream into the original method call and associated parameters (the deserialization of parameters being referred to as unmarshalling). Finally, the skeleton calls the implementation of the method on the server. The stages of this process are shown diagrammatically in Figure 3.13. Even this is a simplification of what is actually happening at the network level, however, since the transport layer and a special layer called the remote reference layer will also be involved at each end of the transmission. In fact, the skeleton was removed entirely in J2SE 1.2 and server programs now communicate directly with the remote reference layer. However, the basic principles remain the same and Figure 3.13 still provides a useful graphical representation of the process.

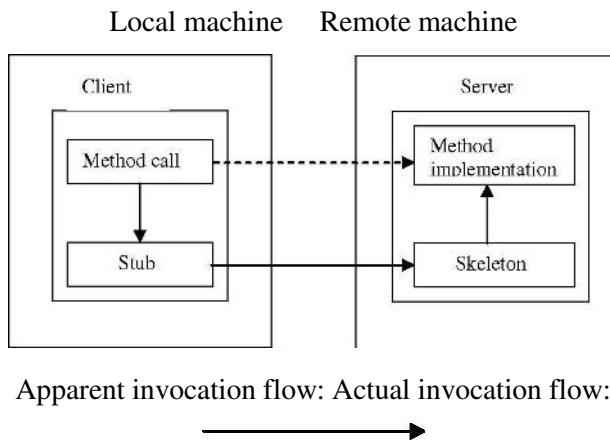


Figure 3.13 Using RMI to invoke a method of a remote object.

If the method has a return value, then the above process is reversed, with the return value being serialised on the server (by the skeleton) and deserialised on the client (by the stub).

#### 3.2.9.2 Java RMI

The Java Remote Method Invocation (RMI) package is a Java-centric scheme for distributed objects that is now a part of the core Java API. RMI offers some of the critical elements of a distributed object system for Java, plus

some other features that are made possible by the fact that RMI is a Java-only system. RMI has object communication facilities that are analogous to CORBA's IIOP, and its object serialization system provides a way for you to transfer or request an object instance by value from one remote process to another.

**Remote Object Interfaces.** Since RMI is a Java-only distributed object scheme, all object interfaces are written in Java. Client stubs and server skeletons are generated from this interface, but using a slightly different process than in CORBA. First, the interface for the remote object has to be written as extending the `java.rmi.Remote` interface. The `Remote` interface doesn't introduce any methods to the object's interface; it just serves to mark remote objects for the RMI system. Also, all methods in the interface must be declared as throwing the `java.rmi.RemoteException`. The `RemoteException` is the base class for many of the exceptions that RMI defines for remote operations, and the RMI engineers decided to expose the exception model in the interfaces of all RMI remote objects. This is one of the drawbacks of RMI: it requires you to alter an existing interface in order to apply it to a distributed environment.

**Server Implementations.** Once the remote object's Java interface is defined, a server implementation of the interface can be written. In addition to implementing the object's interface, the server also typically extends the `java.rmi.server.UnicastRemoteObject` class. `UnicastRemoteObject` is an extension of the `RemoteServer` class, which acts as a base class for server implementations of objects in RMI. Subclasses of `RemoteServer` can implement different kinds of object distribution schemes, like replicated objects, multicast objects, or point-to-point communications. The current version of RMI (1.1) only supports remote objects that use point-to-point communication, and `UnicastRemoteObject` is the only subclass of `RemoteServer` provided. RMI doesn't require your server classes to derive from a `RemoteServer` subclass, but doing so lets your server inherit specialized implementations of some methods from `Object` (`hashCode()`, `equals()`, and `toString()`) so that they do the right thing in a remote object scenario. If you decide that you don't want to subclass from a `RemoteServer` subclass for some reason, then you have to either provide your own special implementations for these methods or live with the fact that these methods may not behave consistently on your remote objects. For example, if you have two client stubs that refer to the same remote object, you would probably want their `hashCode()` methods to return the same value, but the standard `Object` implementation will return independent hash codes for the two stubs. The same inconsistency applies to the standard `equals()` and `toString()` methods.

**The RMI Registry.** In RMI, the registry serves the role of the Object Manager and Naming Service for the distributed object system. The registry runs in its own Java runtime environment on the host that's serving objects. Unlike CORBA, the RMI registry is only required to be running on the server of a remote object. Clients of the object use classes in the RMI package to communicate with the remote registry to look up objects on the server. You start an RMI registry on a host by running the `rmiregistry` command, which is included in the standard JDK distribution. By default, the registry listens to port 1099 on the local host for connections, but you can specify any port for the registry process by using a command-line option:

```
objhost% rmiregistry 4001
```

Once the registry is running on the server, you can register object implementations by name, using the `java.rmi.Naming` interface. We'll see the details of registering server implementations in the next section. A registered class on a host can then be located by a client by using the `lookup()` method on the `Naming` interface. You address remote objects using a URL-like scheme. For example,

```
MyObject obj1 =  
    (MyObject)Naming.lookup("rmi://objhost.myorg.com/Object1");
```

will look up an object registered on the host `objhost.myorg.com` under the name `Object1`. You can have multiple registries running on a server host, and address them independently using their port assignments. For example, if you have two registries running on the `objhost` server, one on port 1099 and another on port 2099, you can locate objects in either registry using URLs that include the port numbers:

```
MyObject obj1 =  
    (MyObject)Naming.lookup("rmi://objhost.myorg.com:1099/Object1");  
MyObject obj2 =  
    (MyObject)Naming.lookup("rmi://objhost.myorg.com:2099/Object2");
```

**Client Stubs and Server Skeletons.** Once you've defined your object's interface and derived a server implementation for the object, you can create a client stub and server skeleton for your object. First the interface and the server implementation are compiled into bytecodes using the `javac` compiler, just like normal classes. Once we have the bytecodes for the interface and the server implementation, we have to generate the linkage from the client through the RMI registry to the object implementation we just generated. This is done using the RMI

stub compiler, rmic. Suppose we've defined a remote interface called MyObject, and we've written a server implementation called MyObjectImpl, and compiled both of these into bytecodes. Assuming that we have the compiled classes in our CLASSPATH, we would generate the RMI stub and skeleton for the class with the rmic compiler:

```
myhost% rmic MyObject
```

The rmic compiler bootstraps off of the Java bytecodes for the object interface and implementation to generate a client stub and a server skeleton for the class. A client stub is returned to a client when a remote instance of the class is requested through the Naminginterface. The stub has hooks into the object serialization subsystem in RMI for marshaling method parameters.

The server skeleton acts as an interface between the RMI registry and instances of the object implementation residing on a host. When a client request for a method invocation on an object is received, the skeleton is called on to extract the serialized parameters and pass them to the object implementation.

**Registering and Using a Remote Object.** Now we have a compiled interface and implementation for our remote object, and we've created the client stub and server skeleton using the rmic compiler. The final hurdle is to register an instance of our implementation on a remote server, and then look up the object on a client. Since RMI is a Java-centric API, we can rely on the bytecodes for the interface, the implementation, the rmic-generated stub, and skeleton being loaded automatically over the network into the Java runtimes at the clients. A server process has to register an instance of the implementation with a RMI registry running on the server:

```
MyObjectImpl obj = new MyObjectImpl();
Naming.rebind("Object1", obj);
```

Once this is done, a client can get a reference to the remote object by connecting to the remote registry and asking for the object by name:

```
System.setSecurityManager(new java.rmi.RMISecurityManager());
MyObject objStub = (MyObject)Naming.lookup("rmi://objhost/Object1");
```

Before loading the remote object stub, we installed a special RMI security manager with the Systemobject. The RMI security manager enforces a security policy for remote stubs to prevent them from doing illicit snooping or sabotage when they're loaded into your local Java environment from a network source. If a client doesn't install an RMI security manager, then stub classes can only be loadable from the local file system.

**Serializing Objects.** Another Java facility that supports RMI is object serialization. The java.io package includes classes that can convert an object into a stream of bytes and reassemble the bytes back into an identical copy of the original object. Using these classes, an object in one process can be serialized and transmitted over a network connection to another process on a remote host. The object (or at least a copy of it) can then be reassembled on the remote host.

An object that you want to serialize has to implement the java.io.Serializable interface. With this done, an object can be written just as easily to a file, a string buffer, or a network socket. For example, assuming that Foo is a class that implements Serializable, the following code writes Foo on an object output stream, which sends it to the underlying I/O stream:

```
Foo myFoo = new Foo();
OutputStream out = ... // Create output stream to object destination
ObjectOutputStream oOut = new ObjectOutputStream(out);
oOut.writeObject(myFoo);
```

The object can be reconstructed just as easily:

```
InputStream in = ... // Create input stream from source of object
ObjectInputStream oIn = new ObjectInputStream(in);
Foo myFoo = (Foo)oIn.readObject();
```

We've simplified things a bit by ignoring any exceptions generated by these code snippets. Note that serializing objects and sending them over a network connection is very different from the functionality provided by the ClassLoader, which we saw earlier in this book. The ClassLoader loads class definitions into the Java runtime, so that new instances of the class can be created. The object serialization facility allows an actual object to be serialized in its entirety, transmitted to any destination, and then reconstructed as a precise replica of the original.

When you serialize an object, all of the objects that it references as data members will also be serialized, and all of their object references will be serialized, and so on. If you attempt to serialize an object that doesn't implement

the `Serializable` interface, or an object that refers to non-serializable objects, then a `NotSerializableException` will be thrown. Method arguments that aren't objects are serialized automatically using their standard byte stream formats.

In RMI, the serialization facility is used to marshal and unmarshal method arguments that are objects, but that are not remote objects. Any object argument to a method on a remote object in RMI must implement the `Serializable` interface, since the argument will be serialized and transmitted to the remote host during the remote method invocation.

### 3.2.9.3 Implementation Details

The packages used in the implementation of an RMI client-server application are `java.rmi`, `java.rmi.server` and `java.rmi.registry`, though only the first two need to be used explicitly. The basic steps are listed below.

- 1 Create the interface.
- 2 Define a class that implements this interface.
- 3 Create the server process.
- 4 Create the client process.

#### Simple Example

This first example application simply displays a greeting to any client that uses the appropriate interface registered with the naming service to invoke the associated method implementation on the server. In a realistic application, there would almost certainly be more methods and those methods would belong to some class (as will be shown in a later example). However, we shall adopt a minimalistic approach until the basic method has been covered. The required steps will be numbered as above...

##### *1. Create the interface.*

This interface should import package `java.rmi` and must extend interface `Remote`, which (like `Serializable`) is a 'tagging' interface that contains no methods. The interface definition for this example must specify the signature for method `getGreeting`, which is to be made available to clients. This method must declare that it throws a `RemoteException`. The contents of this file are shown below.

```
import java.rmi.*;
public interface Hello extends Remote
{
    public String getGreeting() throws RemoteException;
}
```

##### *2. Define a class that implements this interface.*

The implementation file should import packages `java.rmi` and `java.rmi.server`. The implementation class must extend class `RemoteObject` or one of `RemoteObject`'s subclasses. In practice, most implementations extend subclass `UnicastRemoteObject`, since this class supports point-to-point communication using TCP streams. The implementation class must also implement our interface `Hello`, of course, by providing an executable body for the single interface method `getGreeting`. In addition, we must provide a constructor for our implementation object (even if we simply give this constructor an empty body, as below). Like the method(s) declared in the interface, this constructor must declare that it throws a `RemoteException`. Finally, we shall adopt the common convention of appending `Impl` onto the name of our interface to form the name of the implementation class.

```
import java.rmi.*;
import java.rmi.server.*;
public class HelloImpl extends UnicastRemoteObject implements Hello {
    public HelloImpl() throws RemoteException
    {
        //No action needed here.
    }
    public String getGreeting() throws RemoteException
    { return ("Hello there!"); } }
```

##### *3. Create the server process.*

The server creates object(s) of the above implementation class and registers them with a naming service called the

registry. It does this by using static method *rebind* of class *Naming* (from package *java.rmi*). This method takes two arguments:

- a *String* that holds the name of the remote object as a URL with protocol *rmi*;
- a reference to the remote object (as an argument of type *Remote*). The method establishes a connection between the object's name and its reference. Clients will then be able to use the remote object's name to retrieve a reference to that object via the registry.

The URL string, as well as specifying a protocol of *rmi* and a name for the object, specifies the name of the remote object's host machine. For simplicity's sake, we shall use *localhost* (which is what RMI assumes by default anyway). The default port for RMI is 1099, though we can change this to any other convenient port if we wish. The code for our server process is shown below and contains just one method: *main*. To cater for the various types of exception that may be generated, this method declares that it throws *Exception*.

```
import java.rmi.*;
public class HelloServer
{
    private static final String HOST = "localhost";
    public static void main(String[] args) throws Exception
    {
        //Create a reference to an
        //implementation object...
        HelloImpl temp = new HelloImpl();
        //Create the string URL holding the object's name...
        String rmiObjectName = "rmi://" + HOST + "/Hello";
        //(Could omit host name here, since 'localhost' would be assumed by default.)
        //'Bind' the object reference to the name...
        Naming.rebind(rmiObjectName,temp);
        //Display a message so that we know the process has been completed...
        System.out.println("Binding complete...\n");
    }
}
```

#### 4. Create the client process.

The client obtains a reference to the remote object from the registry. It does this by using method *lookup* of class *Naming*, supplying as an argument to this method the same URL that the server did when binding the object reference to the object's name in the registry. Since *lookup* returns a *Remote* reference, this reference must be typecast into an *Hello* reference (not an *HelloImpl* reference!). Once the *Hello* reference has been obtained, it can be used to call the solitary method that was made available in the interface.

```
import java.rmi.*;
public class HelloClient
{
    private static final String HOST = "localhost";
    public static void main(String[] args) {
        try
        {
            //Obtain a reference to the object from the
            //registry and typecast it into the appropriate type...
            Hello greeting = (Hello)Naming.lookup("rmi://" + HOST + "/Hello");
            //Use the above reference to invoke the remote object's method...
            System.out.println("Message received: " + greeting.getGreeting());
        } catch(ConnectException conEx)
        {
            System.out.println("Unable to connect to server!"); System.exit(1);
        } catch(Exception ex) { ex.printStackTrace(); System.exit(1); } }
}
```

Note that some authors choose to combine the implementation and server into one class. This author, however, feels that the separation of the two probably results in a clearer delineation of responsibilities.

## Compilation and Execution

There are several steps that need to be carried out, as described below.

### 1. Compile all files with javac.

This is straightforward...

```
javac Hello.java
```

```
javac HelloImpl.java  
javac HelloServer.java  
javac HelloClient.java
```

## 2. Compile the implementation class with the rmic compiler.

This compiler is one of the utilities supplied with the J2SE. Though it might seem strange to have a compiler operate on anything other than source code, this compiler operates on the .class file generated by the above compilation of the implementation file. Used without any command line option, it will generate both a stub file and a skeleton file. As mentioned in Section 5.1, however, Java 2 does not require the skeleton file. If Java 2 is being used, then command line option -v1.2 should be employed (as shown below), so that only the stub file is generated.

```
rmic -v1.2 HelloImpl
```

This will cause a file with the name *HelloImpl\_stub.class* to be created.

## 3. Start the RMI registry.

Enter the following command:

```
rmiregistry
```

When this is executed, the only indication that anything has happened is a change in the command window's title.

## 4. Open a new window and run the server.

From the new window, invoke the Java interpreter:

```
java HelloServer
```

## 5. Open a third window and run the client.

Again, invoke the Java interpreter:

```
java HelloClient
```

Since the server process and the RMI registry will continue to run indefinitely after the client process has finished, they will need to be closed down by entering Ctrl-C in each of their windows. Now that the basic process has been covered, the next section will examine a more realistic application of RMI.

### 3.2.9.4 Using RMI Meaningfully

In a realistic RMI application, multiple methods and probably multiple objects will be employed. With such real-world applications, there are two possible strategies that may be adopted, as described below.

- Use a single instance of the implementation class to hold instance(s) of a class whose methods are to be called remotely. Pass instance(s) of the latter class as argument(s) of the constructor for the implementation class.
- Use the implementation class directly for storing required data and methods, creating instances of this class, rather than using separate class(es).

Some authors use the first strategy, while others use the second. Each approach has its merits and both will be illustrated below by implementing the same application, so that the reader may compare the two techniques and choose his/her own preference.

Example This application will make bank account objects available to connecting clients, which may then manipulate these remote objects by invoking their methods. For simplicity's sake, just four account objects will be created and the practical considerations relating to security of such accounts will be ignored completely!

Each of the above two methods will be implemented in turn.

#### **Method 1**

Instance variables and associated methods for an individual account will be encapsulated within an application class called *Account*. If this class does not already exist, then it must be created, adding a further step to the four

steps specified in Section 5.2. This step will be inserted as step 3 in the description below.

### 1. Create the interface.

Our interface will be called *Bank1* and will provide access to details of all accounts via method *getBankAccounts*. This method returns a *Vector* of *Account* objects that will be declared within the implementation class. The code is shown below:

```
import java.rmi.*;
import java.util.Vector;
public interface Bank1 extends Remote
{
    public Vector<Account> getBankAccounts() throws RemoteException;
}
```

### 2. Define the implementation.

The code for the implementation class provides both a definition for the above method and the definition for a constructor to set up the *Vector* of *Account* objects:

```
import java.rmi.*;
import java.rmi.server.*;
import java.util.Vector;
public class Bank1Impl extends UnicastRemoteObject implements Bank1
{
    //Declare the Vector that will hold Account objects...
    private Vector<Account> acctInfo;
    //The constructor must be supplied with a Vector of //Account objects...
    public Bank1Impl(Vector<Account> acctVals) throws RemoteException {
        acctInfo = acctVals; }
    //Definition for the single interface method...
    public Vector<Account> getBankAccounts()
    throws RemoteException
    {
        return acctInfo;
    }
}
```

### 3. Create any required application classes.

In this example, there is only class *Account* to be defined. Since it is to be used in the return value for our interface method, it must be declared to implement the *Serializable* interface (contained in package *java.io*).

```
public class Account implements java.io.Serializable
{
    //Instance variables...
    private int acctNum;
    private String surname;
    private String firstNames;
    private double balance;
    //Constructor...
    public Account(int acctNo, String sname, String fnames, double bal)
    {
        acctNum = acctNo;
        surname = sname;
        firstNames = fnames;
        balance = bal;
    }
    //Methods...
    public int getAcctNum() { return acctNum; }
    public String getName() { return (firstNames + " " + surname); }
    public double getBalance() { return balance; }
    public double withdraw(double amount) {
        if ((amount > 0) && (amount <= balance)) return amount; else return 0; }
    public void deposit(double amount) { if (amount > 0) balance += amount; }
}
```

#### 4. Create the server process.

The code for the server class sets up a *Vector* holding four initialised *Account* objects and then creates an implementation object, using the *Vector* as the argument of the constructor. The reference to this object is bound to the programmer-chosen name *Accounts* (which must be specified as part of a URL identifying the host machine) and placed in the registry. The server code is shown below.

```
import java.rmi.*;
import java.util.Vector;
public class Bank1Server
{
private static final String HOST = "localhost";
public static void main(String[] args) throws Exception
{ //Create an initialised array of four Account objects...
Account[] account =
{ new Account(111111,"Smith","Fred James",112.58),
  new Account(222222,"Jones","Sally",507.85),
  new Account(234567,"White","Mary Jane",2345.00),
  new Account(666666,"Satan","Beelzebub",666.00)};
Vector<Account> acctDetails = new Vector<Account>();
//Insert the Account objects into the Vector...
for (int i=0; i<account.length; i++) acctDetails.add(account[i]);
//Create an implementation object, passing the
//above Vector to the constructor...
Bank1Impl temp = new Bank1Impl(acctDetails);
//Save the object's name in a String...
String rmiObjectName = "rmi://" + HOST + "/Accounts";
//(Could omit host name, since 'localhost' would be assumed by default.)
//Bind the object's name to its reference...
Naming.rebind(rmiObjectName,temp);
System.out.println("Binding complete...\n");
}
}
```

#### 5. Create the client process.

The client uses method *lookup* of class *Naming* to obtain a reference to the remote object, typecasting it into type *Bank1*. Once the reference has been retrieved, it can be used to execute remote method *getBankAccounts*. This returns a reference to the *Vector* of *Account* objects which, in turn, provides access to the individual *Account* objects. The methods of these *Account* objects can then be invoked as though those objects were local.

```
import java.rmi.*;
import java.util.Vector;
public class Bank1Client
{
private static final String HOST =
"localhost";
public static void main(String[] args)
{
try
{
//Obtain a reference to the object from the
//registry and typecast it into the appropriate
//type...
Bank1 temp = (Bank1)Naming.lookup("rmi://" + HOST + "/Accounts");
Vector<Account> acctDetails = temp.getBankAccounts();
//Simply display all acct details...
for (int i=0; i<acctDetails.size(); i++)
{
//Retrieve an Account object from the Vector...
Account acct = acctDetails.elementAt(i);
//Now invoke methods of Account object to display its details...
System.out.println("\nAccount number: " + acct.getAcctNum());
System.out.println("Name: " + acct.getName());
System.out.println("Balance: " + acct.getBalance());
}
}
}
```

```

        }
    }
    catch(ConnectException conEx)
    {
        System.out.println( "Unable to connect to server!" );
        System.exit(1);
    }
    catch(Exception ex) { ex.printStackTrace(); System.exit(1); } } }
```

The steps for compilation and execution are the same as those outlined in the previous section for the *Hello* example, with the minor addition of compiling the source code for class *Account*. The steps are shown below.

**1. Compile all files with javac.**

This time, there are five files...

```

javac Bank1.java
javac Bank1Impl.java
javac Account.java
javac Bank1Server.java
javac Bank1Client.java
```

**2. Compile the implementation class with the rmic compiler.**

```
rmic -v1.2 Bank1Impl
```

This will cause a file with the name *Bank1Impl\_stub.class* to be created.

**3. Start the RMI registry.**

Enter the following command:

```
rmiregistry
```

**4. Open a new window and run the server.**

From the new window, invoke the Java interpreter:

```
java Bank1Server
```

**5. Open a third window and run the client.**

Again, invoke the Java interpreter:

```
java Bank1Client
```

Once again, the server process and the RMI registry will need to be closed down by entering Ctrl-C in each of their windows.

### **Method 2**

For this method, no separate *Account* class is used. Instead, the data and methods associated with an individual account will be defined directly in the implementation class. The interface will make the methods available to client processes. The same four steps as were identified in Section 5.2 must be carried out, as described below.

**1. Create the interface.**

The same five methods that appeared in class *Account* in *Method 1* are declared, but with each now declaring that it throws a *RemoteException*.

```

import java.rmi.*;

public interface Bank2 extends Remote
{
    public int getAcctNum()throws RemoteException;
    public String getName()throws RemoteException;
    public double getBalance()throws RemoteException;
    public double withdraw(double amount) throws RemoteException;
    public void deposit(double amount) throws RemoteException; }
```

**2. Define the implementation.**

As well as holding the data and method implementations associated with an individual account, this class defines a constructor for implementation objects. The method definitions will be identical to those that were previously held within the *Account* class, of course.

```
import java.rmi.*;
import java.rmi.server.*;
public class Bank2Impl extends UnicastRemoteObject implements Bank2
{
    private int acctNum;
    private String surname;
    private String firstNames;
    private double balance;
    //Constructor for implementation objects...
    public Bank2Impl(int acctNo, String sname,
                     String fnames, double bal) throws RemoteException
    { acctNum = acctNo; surname = sname; firstNames = fnames; balance = bal; }
    public int getAcctNum() throws RemoteException { return acctNum; }
    public String getName() throws RemoteException {
        return (firstNames + " " + surname); }
    public double getBalance() throws RemoteException { return balance; }
    public double withdraw(double amount) throws RemoteException
    { if ((amount>0) && (amount<=balance)) return amount; else return 0; }
    public void deposit(double amount) throws RemoteException
    { if (amount > 0) balance += amount; }
}
```

### 3. Create the server process.

The server class creates an array of implementation objects and binds each one individually to the registry. The name used for each object will be formed from concatenating the associated account number onto the word 'Account' (forming 'Account111111', etc.).

```
import java.rmi.*;
public class Bank2Server
{
    private static final String HOST = "localhost";
    public static void main(String[] args) throws Exception
    {
        //Create array of initialised implementation
        //objects...
        Bank2Impl[] account =
            {new Bank2Impl(111111,"Smith", "Fred James",112.58),
             new Bank2Impl(222222,"Jones", "Sally",507.85),
             new Bank2Impl(234567,"White", "Mary Jane",2345.00),
             new Bank2Impl(666666,"Satan", "Beelzebub",666.00)};

        for (int i=0; i<account.length; i++)
        {
            int acctNum = account[i].getAcctNum();
            /* Generate each account name (as a concatenation of 'Account' and the
               account number) and bind it to the appropriate object reference in the
               array... */
            Naming.rebind("rmi://" + HOST + "/Account" + acctNum, account[i]);
        }
        System.out.println("Binding complete...\n");
    }
}
```

### 4. Create the client process.

The client again uses method *lookup*, this time to obtain references to individual accounts (held in separate implementation objects):

```
import java.rmi.*;
public class Bank2Client
```

```

{
    private static final String HOST = "localhost";
    private static final int[] acctNum = {111111,222222,234567,666666};
    public static void main(String[] args)
    {
        try
        {
            //Simply display all account details...
            for (int i=0; i<acctNum.length; i++)
            {
                /* Obtain a reference to the object from the
                   registry and typecast it into the appropriate type... */
                Bank2 temp =
                    (Bank2)Naming.lookup("rmi://" + HOST + "/Account" + acctNum[i]);
                //Now invoke the methods of the interface to
                //display details of associated account...
                System.out.println("\nAccount number: " + temp.getAcctNum());
                System.out.println("Name: " + temp.getName());
                System.out.println("Balance: " + temp.getBalance());
            }
        }
        catch(ConnectException conEx)
        {
            System.out.println( "Unable to connect to server!");
            System.exit(1);
        }
        catch(Exception ex)
        {
            ex.printStackTrace();
            System.exit(1); } } }
}

```

### 3.2.9.5 Example of a RMI Solver

Now let's go back to our Solver example and distribute it using RMI. First, we would have to rewrite the Solver interface so that it implements the `java.rmi.Remote` interface. The methods on the interface also have to be specified as throwing `RemoteExceptions`. This modified version of the Solver interface, the `RMISolver`, is shown in Example:

#### Example 3.14. Interface for a RMI Solver

```

package dcj.examples.rmi;
import java.rmi.*;
import java.io.OutputStream;
public interface RMISolver extends java.rmi.Remote
{
    public boolean solve() throws RemoteException;
    public boolean solve(RMIProblemSet s,
                        int numIters) throws RemoteException;
    public RMIProblemSet getProblem() throws RemoteException;
    public boolean setProblem(RMIProblemSet s) throws RemoteException;
    public int getIterations() throws RemoteException;
    public boolean setIterations(int numIter) throws RemoteException;
}

```

There are two methods in this interface, the `solve()` method with arguments, and the `setProblem()` method, where we have problem set arguments that we want to pass into the remote method invocation. We could achieve this by creating a version of the `ProblemSet` class that implements the `Serializable` interface. If we did that, the problem set would be sent to the remote host by value—the remote object would be operating on a copy of the problem set. But in both of these cases we want to pass the problem set by reference; we want the remote Solver to operate on the same problem set object that we have on the client, so that when the solution is stored in the problem set, we will see it automatically on the client. We can do this in RMI by making a remote version of the `ProblemSet` class. With an RMI-enabled `ProblemSet` interface, we can use an instance of an implementation of the interface as an argument to remote methods, and the remote object will receive a stub to the local `ProblemSet`. The RMI

version of the ProblemSet interface, the RMIProblemSet, is shown in Example 3.15.

### Example 3.15. Interface for an RMI ProblemSet

```
package dcj.examples.rmi;
import java.rmi.*;

public interface RMIProblemSet extends Remote {
    public double getValue() throws RemoteException;
    public double getSolution() throws RemoteException;
    public void setValue(double v) throws RemoteException;
    public void setSolution(double s) throws RemoteException;
}
```

Now we'll need to write server-side implementations of these interfaces. Our server-side implementation of the RMISolver derives from `java.rmi.UnicastRemoteObject`, and is shown in Example 3.16. The implementation of the RMIProblemSet interface is shown in Example 3.16. It also extends the `UnicastRemoteObject` class.

### Example 3.16. Implementation of the RMISolver

```
package dcj.examples.rmi;

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;
import java.io.*;

public class RMISolverImpl extends UnicastRemoteObject implements RMISolver {
    // Protected implementation variables
    protected int numIterations = 1; // not used for this Solver...
    protected RMIProblemSet currProblem = null;

    // Constructors
    public RMISolverImpl() throws RemoteException { super(); }
    public RMISolverImpl(int numIter) throws RemoteException {
        super();
        numIterations = numIter;
    }

    // Public methods
    public boolean solve() throws RemoteException {
        System.out.println("Solving current problem...");
        return solve(currProblem, numIterations);
    }

    public boolean solve(RMIProblemSet s, int numIters) throws RemoteException {
        boolean success = true;
        if (s == null) {
            System.out.println("No problem to solve.");
            return false;
        }
        System.out.println("Problem value = " + s.getValue());

        // Solve problem here...
        try {
            s.setSolution(Math.sqrt(s.getValue()));
        }
        catch (ArithmaticException e) {
            System.out.println("Badly-formed problem.");
            success = false;
        }
        System.out.println("Problem solution = " + s.getSolution());
        return success;
    }

    public RMIProblemSet getProblem() throws RemoteException {
        return currProblem;
    }

    public boolean setProblem(RMIProblemSet s) throws RemoteException {
```

```

currProblem = s;
return true;
}
public int getIterations() throws RemoteException {
    return numIterations;
}
public boolean setIterations(int numIter) throws RemoteException {
    numIterations = numIter;
    return true;
}
public static void main(String argv[]) {
    try {
        // Register an instance of RMISolverImpl with the
        // RMI Naming service
        String name = "TheSolver";
        System.out.println("Registering RMISolverImpl as \\" + name + "\\");
        RMISolverImpl solver = new RMISolverImpl();
        Naming.rebind(name, solver);
        System.out.println("Remote Solver ready...");
    }
    catch (Exception e) {
        System.out.println("Caught exception while registering: " + e);
    }
}
}
}

```

### **Example 3.17. Implementation of the RMIProblemSet**

```

package dcj.examples.rmi;

import java.rmi.*;
import java.rmi.server.UnicastRemoteObject;

public class RMIProblemSetImpl
    extends java.rmi.server.UnicastRemoteObject
    implements RMIProblemSet {
    protected double value;
    protected double solution;

    public RMIProblemSetImpl() throws RemoteException {
        value = 0.0;
        solution = 0.0;
    }
    public double getValue() throws RemoteException {
        return value;
    }
    public double getSolution() throws RemoteException {
        return solution;
    }
    public void setValue(double v) throws RemoteException {
        value = v;
    }
    public void setSolution(double s) throws RemoteException {
        solution = s;
    }
}

```

These implementations of our Solver and ProblemSet interfaces are very similar to those that we created for the earlier CORBA examples. As in our earlier examples, the Solver simply performs a square root on the ProblemSet floating-point value. The RMISolverImpl has a main() method that registers a RMISolverImpl object with the local RMI registry.

Now we compile our interfaces and our server implementations into bytecodes, then generate their client stubs and server skeletons using the rmic compiler:

```
myhost% rmic dcj.examples.rmi.RMIProblemSetImpl
myhost% rmic dcj.examples.rmi.RMISolverImpl
```

The last required item is a client to use our remote object. The RMISolverClient in previous example is a simple client for the remote solver. The client has a single main() method where it gets a stub for the remote solver and asks it to solve a problem. The first line of the main() method installs the RMISecurityManager. Next, the client looks up the solver registered on the remote server through the Naming.lookup() method. Once it has the RMISolver stub, it creates a RMIProblemSetImpl object (our RMI-enabled ProblemSet implementation), and passes it into the solver's solve() method. The remote solver receives a stub to the RMIProblemSetImpl object on the client host, and solves the problem it represents. The methods that the remote RMISolver calls on the RMIProblemSet stub are invoked remotely on the RMIProblemSetImpl object on the client. Once the solve() method returns, our client can get the problem solution from the RMIProblemSetImpl object that it passed into the remote method call.

### **Example 3.18. An RMISolver Client**

```
package dcj.examples.rmi;

import java.rmi.*;
import java.rmi.server.*;

public class RMISolverClient {

    public static void main(String argv[]) {
        // Install a security manager that can handle remote stubs
        System.setSecurityManager(new RMISecurityManager());

        // Get a remote reference to the RMISolver class
        String name = "rmi://objhost.myorg.com/TheSolver";
        System.out.println("Looking up " + name + "...");
        RMISolver solver = null;
        try {
            solver = (RMISolver)Naming.lookup(name);
        }
        catch (Exception e) {
            System.out.println("Caught an exception looking up Solver.");
            System.exit(1);
        }

        // Make a problem set for the solver
        RMIProblemSetImpl s = null;
        try {
            s = new RMIProblemSetImpl();
            s.setValue(Double.valueOf(argv[0]).doubleValue());
        }
        catch (Exception e) {
            System.out.println("Caught exception initializing problem.");
            e.printStackTrace();
        }

        // Ask solver to solve
        try {
            if (solver.solve(s, 1)) {
                System.out.println("Solver returned solution: " + s.getSolution());
            }
            else {
                System.out.println(
                    "Solver was unable to solve problem with value = " + s.getValue());
            }
        }
        catch (RemoteException e) {
            System.out.println("Caught remote exception.");
            System.exit(1);
        }
    }
}
```

Finally, we're ready to try our distributed object system. First, we start a registry on the host that is serving objects through the Naming service:

```
objhost% rmiregistry &
```

Now we can register a RMISolverImpl object by running the main() method on the RMISolverImpl class:

```
objhost% java dcj.examples.rmi.RMISolverImpl  
Registering RMISolverImpl as "TheSolver"  
Remote Solver ready...
```

Back on our client host, we can run the client class:

```
client% java dcj.examples.rmi.RMISolverClient 47.0  
Looking up "rmi://objhost.myorg.com/TheSolver"...
Solver returned solution: 6.855654600401044
```

It's important to note here that the ProblemSet we're sending to the remote Solver object through a remote method call isn't being served in the same way as the Solver object. The Solver server doesn't need to lookup the ProblemSet object through the RMI registry. A stub interface to the client-side RMIProblemSetImpl object is automatically generated on the server side by the underlying RMI system.

### 3.1.9.6 RMI Security

If both the client and server processes have direct access to the same class files, then there is no need to take special security precautions, since no security holes can be opened up by such an arrangement. However, an application receiving an object for which it does not have the corresponding class file can try to load that class file from a remote location and instantiate the object in its JVM. Unfortunately, an object passed as an RMI argument from such a remote source can attempt to initiate execution on the client's machine immediately upon deserialisation without the user/programmer doing anything with it! Such a security breach is not permitted to occur, of course. The loading of this file is handled by an object of class *SecureClassLoader*, which must have security restrictions defined for it. File *java.policy* defines these security restrictions, while file *java.security* defines the security properties. Implementation of the security policy is controlled by an object of class *RMISecurityManager* (a subclass of *SecurityManager*). The *RMISecurityManager* creates the same 'sandbox' rules that govern applets. Without such an object, a Java application will not even attempt to load classes that are not from its local file system.

Though the security policy can be modified by use of the Java utility *policytool*, this can be done only for individual hosts, so it is probably more straightforward to write and install one's own security manager. There is a default *RMISecurityManager*, but this relies on the system's default security policy, which is far too restrictive to permit the downloading of class files from a remote site. In order to get round this problem, we must create our own security manager that extends *RMISecurityManager*. This security manager must provide a definition for method *checkPermission*, which takes a single argument of class *Permission* from package *java.security*. For simplicity's sake and because the complications involved with specifying security policies go beyond the scope of this text, we shall illustrate the procedure with the simplest possible security manager – one that allows everything! The code for this security manager is shown below.

```
import java.rmi.*;
import java.security.*;
public class ZeroSecurityManager extends RMISecurityManager
{
    public void checkPermission(Permission permission)
    {
        System.out.println("checkPermission for : " + permission.toString());
    }
}
```

As with all our associated RMI application files, this file must be compiled with *javac*. The client program must install an object of this class by invoking method *setSecurityManager*, which is a static method of class *System* that takes a single argument of class *SecurityManager* (or a subclass of *SecurityManager*, of course). For illustration purposes, the code for our *HelloClient* program is reproduced below, now incorporating a call to *setSecurityManager*. This call is shown in emboldened text.

```
import java.rmi.*;
public class HelloClient
{
```

```

private static final String HOST = "localhost";
public static void main(String[] args)
{
    //Here's the new code...
    if (System.getSecurityManager() == null)
    {
        System.setSecurityManager(new ZeroSecurityManager());
    }
    try
    {
        Hello greeting = (Hello)Naming.lookup( "rmi://" + HOST + "/Hello");
        System.out.println("Message received: " + greeting.getGreeting());
    }
    catch(ConnectException conEx)
    {
        System.out.println( "Unable to connect to server!");
        System.exit(1);
    }
    catch(Exception ex) { ex.printStackTrace(); System.exit(1); } } }

```

When executing the server, we need to specify where the required *.class* files are located, so that clients may download them. To do this, we need to set the *java.rmi.server.codebase* property to the URL of this location, at the same time that the server is started. This is achieved by using the *-D* command line option to specify the setting of the codebase property. For example, if the URL of the file location is *http://java.shu.ac.uk/rmi/*, then the following line would set our *HelloServer* program running and would set the codebase property to the required location at the same time:

```
java -Djava.rmi.server.codebase=http://java.shu.ac.uk/rmi/ HelloServer
```

It is very easy to make a slip during the above process that will cause the application to fail, but coverage of these problems goes beyond the scope of this text.

### 3.2.9.7 Exercises

1 Using class *Result* (shown below) and making the minor modification that will ensure that objects of this class are serialisable, make method *getResults* available via an RMI interface. This method should return a *Vector* containing initialised *Result* objects that are set up by a server program (also to be written by you) and made available via an implementation object placed in the RMI registry by the server. The server should store two *Result* objects in the *Vector* contained within the implementation object. Access this implementation object via a client program and use the methods of the *Result* class to display the surname and examination mark for each of the two *Result* objects. You should find the solution to the above problem relatively straightforward by simply modifying the code for the *Bank* example application from this chapter.

```

class Result implements java.io.Serializable
{
    private String surname;
    private int mark;
    public Result(String name, int score) { surname = name; mark = score; }
    public String getName() { return surname; }
    public void setName(String name) { surname = name; }
    public int getMark() { return mark; }
    public void setMark(int score)
    { if ((score>=0) && (score<=100)) mark = score;
}
}

```

2 Repeat the above exercise, this time without using a separate *Result* class, but holding the result methods directly in the implementation class. Store the implementation objects remotely under the names *result1* and *result2*. Access these objects via a client program and use the methods of the implementation class to display the surnames and examination marks for each of the two objects.

### **3.2.10 URLConnections and ClassLoader**

The `java.net` package, in addition to object-oriented representations of IP sockets, also provides objects that support the HTTP protocol for accessing data in the form of addressable documents. HTTP is really an extension of the underlying IP protocol we discussed earlier, designed specifically to provide a way to address different kinds of documents, or pieces of data, distributed on the network. In the rest of this book, we'll see numerous examples of distributed applications whose agents use customized or standard communications protocols to talk to each other. If there is an HTTP server "agent" available on one of the hosts in our distributed application, then we can use the classes discussed in this section to ask it for data documents using the standard HTTP protocol.

To address a specific document or data object, we use a Uniform Resource Locator (URL), which includes four address elements: the protocol, host, port, and document. The Java representation for a URL is the `URL` class, which is constructed with a given protocol, host, port, and document filename. Once the `URL` object is constructed, it allows the user to make the necessary requests to connect to the HTTP server of the data object, query for information about the object, and download the object. The content of the object can be accessed using the `getContent()`, `openConnection()`, or `openStream()` methods on the `URL` object. Of these three methods, `openStream()` is simplest. The `openStream()` method returns an `InputStream` that can be used to read the data contents directly.

When you call `openConnection()` on a `URL` object, you get a `URLConnection` in return. You can use the `URLConnection` to query the data connection's header information for the data object's length, the type of data it contains, the data encoding, etc. You can also control aspects of the data connection that determine when the data object can be pulled from a local cache, whether input or output is to be done over the data connection, and when unmodified data should be read from the server.

The `getContent()` method downloads the data object and returns an `Object` containing the data. Using this method relies upon having a content handler that supports the object's data format and is capable of converting it into a Java object. The `java.net` package allows you to extend the available content handlers using the `ContentHandler` and `ContentHandlerFactory` classes. A `ContentHandler` accepts a `URLConnection`, reads the data from the associated data object, and constructs an appropriate `Object` instance to represent the data object in the Java environment. It is the job of the system-wide `ContentHandlerFactory` to associate the proper `ContentHandler` with each data object referenced by a `URL`. When `getContent()` is called on a `URL` or `URLConnection` object, the `ContentHandlerFactory` is queried for a `ContentHandler` that can read the format of the data at the other end of the connection. The `ContentHandlerFactory` checks the MIME type and encoding of the data object, and returns a `ContentHandler` for that MIME type. The `ContentHandler` that's returned is then asked for an `Object` representing the data by calling its `getContent()` method with the `URLConnection`. Typically, the `ContentHandler` reads the raw data from the `URLConnection`'s `InputStream`, formats the data into an appropriate object representation, and returns the object to the caller.

Suppose we want to connect to an HTTP server containing computational fluid dynamics (CFD) data files stored in a proprietary format. Suppose these data files have a ".cf" suffix, and we decide to reserve the MIME type "application/cfd" for these data files. Now, assuming that the HTTP server has been properly configured to export this MIME type in the content headers it transmits, we can use Java's HTTP support to access these data files from our application by creating our own `ContentHandler` subclass that is capable of reading the data stream and converting it to an appropriate Java object. Example JDC.E11 shows a `CFDContentHandler` that does just this. Its `getContent()` method creates a `CFDDataset` object from the data read from the input stream of the `URLConnection` argument. It assumes that the incoming data is of the expected type and format for the `CFDDataset`; a more robust implementation would check the MIME type of the `URLConnection` and warn the user if the type doesn't match.

#### **Example 3.19. A ContentHandler for CFD File**

```
import java.net.*;
import dcj.examples.Networking.CFDDataset;

public class CFDContentHandler extends ContentHandler {
    public Object getContent(URLConnection u) {
        CFDDataset d = new CFDDataset();
        try {
            InputStream in = u.getInputStream();
```

```

        byte[] buffer = new byte[1024];
        while (in.read(buffer) > 0) {
            d.addData(buffer);
        }
    }
    catch (Exception e) {
        e.printStackTrace();
    }
    return d;
}
}

```

To use our CFDContentHandler to read CFD files, we still need to register a new ContentHandlerFactory that knows about the CFDContentHandler. The CFDContentHandlerFactor in Example 3.20 creates CFDContentHandlers for the application/cfd MIME type. It ignores any other MIME types, but we could also implement it with a reference to a default ContentHandlerFactory that can handle other MIME types.

### **Example 3.20. A Specialized ContentHandlerFactory for CFD Data Files**

```

package dcj.examples.Networking;

import java.net.*;

public class CFDContentHandlerFactory
    implements ContentHandlerFactory {
    public ContentHandler createContentHandler(String mimetype) {
        if (mimetype.compareTo("application/cfd") == 0) {
            return new CFDContentHandler();
        }
        else
            return null;
    }
}

```

Finally, our application can read CFD data files from an HTTP server by first registering the specialized ContentHandlerFactory, and then requesting a CFD file from the HTTP server on which it lives:

```

URLConnection.setContentHandlerFactory(new
CFDContentHandlerFactory());
URL cfdURL = new URL("http://my.data.server/datasets/bigset.cfd");
CFDDataset data = (CFDDataset)cfdURL.getContent();

```

#### **3.2.10.1 When and Where Are URLs Practical?**

We can transmit data around a distributed system using sockets and streams. This method has the advantage of being efficient, since we are using basic IP sockets with minimal protocol overhead getting between us and our data. The downside is that it is our responsibility to know the type and format of the data we're transmitting and receiving. The communication protocol must be mutually agreed upon by all participating computing agents, or we have to establish our own means for communicating metadata about the kind of information with which we are dealing.

Java's HTTP support classes, on the other hand, provide a standard means for serving and accessing data objects, and for easily identifying the type and format of these objects. To make a piece of data available from a URL, we need to install it in the content section of an HTTP server, and configure the server to transmit the appropriate MIME type when the data is accessed. On the receiving end, we simply need to use the data object's URL to access the document, ask the corresponding URLConnection for the type and encoding of the data, and respond accordingly. The downside is that HTTP imposes plenty of protocol overhead on the data stream, which reduces our net data bandwidth between computing agents. Our data is now sharing space in network packets with IP protocol *and* HTTP protocol. Another downside is the relatively basic and simplistic resource naming facility that HTTP provides, compared to formal directory naming services like NIS and LDAP. The simple conclusion is that, for distributed applications that are severely bandwidth-limited, or that need to support complicated resource hierarchies, using the HTTP protocol to access data is probably not the appropriate method. On the other hand, if

you have the luxury of some extra communications bandwidth, and the CPU time to use it, and your resource groupings are relatively simple, then using URLs to access data is a possibility you should consider.

### 3.2.10.2 The ClassLoader

The Java runtime environment is based upon a virtual machine that interprets, verifies, and executes classes in the form of platform-independent bytecodes. In addition, the Java API includes a mechanism for you to load class definitions in their bytecode form, and integrate them into the runtime environment so that instances of the classes can be constructed and used. When your Java files are compiled, a similar mechanism is invoked whenever an import statement is encountered. The referenced class or package of classes is loaded from files in bytecode format, using the CLASSPATH environment variable to locate them on the local file system.

In addition to this default policy for loading classes, the java.lang.ClassLoader class allows the user to define custom policies and mechanisms for locating and loading classes into the runtime environment. The ClassLoader is an abstract class. Subclasses must define an implementation for the loadClass() method, which is responsible for locating the class based upon the given string name, loading the bytecodes comprising the class definition, and (optionally) resolving the class. A class has to be resolved before it can be constructed or before any of its methods can be called. Resolving a class includes finding all of the other classes that it depends on, and loading them into the runtime as well.

The ClassLoader is an important element of the network support in the Java API. It's used as the basis for supporting Java applets in most Java-enabled web browsers, for example. When an HTML page includes an APPLET tag that references a Java class on the HTTP server, a ClassLoader instance within the browser's Java runtime is used to load the bytecodes of the class into the virtual machine, create an instance of the class, and then execute methods on the new object. Note that this is different from the concept of distributing *objects* using RMI or CORBA. Rather than creating an object on one host and allowing a process on a remote host to call methods on that object, the ClassLoader lets an agent read the bytecodes making up a class definition, and then create an object within its own process. In the rest of this section we'll look at how we can directly use the ClassLoader interface to distribute *classes* in a network environment.

### 3.2.10.3 Loading Classes from the Network

Now, in looking at the overall object model defined by the Java API, we can think of the java.lang.ClassLoader class as an abstract interface for the loading of classes into the runtime environment, and the java.io.InputStream class as the basis for loading data into the runtime environment from different sources and in different formats. An obvious next step would seem to be to put them together, and form the basis for loading classes from all of the sources accessible from subclasses of InputStream. So that's just what we've done, and the result is the StreamClassLoader shown in Example 3.21.

#### Example 3.21. A Network ClassLoader

```
package dcj.util;

import java.lang.*;
import java.net.*;
import java.io.*;
import java.util.Hashtable;

public abstract class StreamClassLoader extends ClassLoader
{

    // Instance variables and default initializations
    Hashtable classCache = new Hashtable();

    InputStream source = null;
    // Constructor
    public StreamClassLoader()
    {
    }

    // Parse a class name from a class locator (URL, filename, etc.)
    protected abstract String parseClassName(String classLoc)
        throws ClassNotFoundException;
```

```

// Initialize the input stream from a class locator
protected abstract void initStream(String classLoc)
throws IOException;

// Read a class from the input stream
protected abstract Class readClass(String classLoc, String className)
throws IOException, ClassNotFoundException;

// Implement the ClassLoader loadClass() method.
// First argument is now interpreted as a class locator, rather than
// simply a class name.
public Class loadClass(String classLoc, boolean resolve)
throws ClassNotFoundException
{
    String className = parseClassName(classLoc);
    Class c = (Class)classCache.get(className);

    // If class is not in cache...
    if (c == null)
    {
        // ...try initializing our stream to its location
        try { initStream(classLoc); }
        catch (IOException e)
        {
            throw new ClassNotFoundException(
                "Failed opening stream to URL.");
        }

        // Read the class from the input stream
        try { c = readClass(classLoc, className); }
        catch (IOException e)
        {
            throw new ClassNotFoundException(
                "Failed reading class from stream: " + e);
        }
    }

    // Add the new class to the cache for the next reference.
    // Note that we cache based on the class name, not locator.
    classCache.put(className, c);
    // Resolve the class, if requested.
    if (resolve)
        resolveClass(c);
    return c;
}
}

```

The abstract StreamClassLoader class provides a generic interface for implementing and using stream-based class loaders. It accomplishes this in part by changing the semantics of the string argument to the loadClass() method on ClassLoader. Whereas ClassLoader defines this argument as the name of the class being sought, the StreamClassLoader broadens the definition to include class "locators" in general. A class locator may be a URL, a host/port/filename combination, or some other means for addressing a class located on the network, or anywhere else accessible via an input stream. Subclasses of StreamClassLoader must define the class locator format they expect, by implementing the parseClassName() method.

The other element of the StreamClassLoader framework is an implementation of loadClass() which allows subclasses to initialize and read their input streams to bring the requested class into the local environment. If the class locator string is successfully parsed by parseClassName(), then the StreamClassLoader calls initStream(), passing the class locator. This method should attempt to initialize the stream to the class specified by the locator. If successful, the StreamClassLoader next calls its readClass() method, passing the class locator and class name. This returns the newly constructed Class object, which is then optionally resolved and returned to the caller.

To demonstrate a practical extension of the StreamClassLoader, Example 3.22 shows a URLClassLoader, which loads classes that are located at URLs on HTTP servers. In this case, a class locator is expected to be in the form of a valid URL. The URLClassLoader utilizes the URL andURLConnection classes to implement the parseClassName(), initStream(), and readClass() methods, as you might expect.

### Example 3.22. A URL-based ClassLoader

```
package dcj.util;
import java.lang.*;
import java.net.*;
import java.io.*;
import java.util.Hashtable;

public class URLClassLoader extends StreamClassLoader
{
    URL classURL = null;
    InputStream classStream = null;

    protected String parseClassName(String classLoc)
        throws ClassNotFoundException
    {
        String className = null;
        // Try constructing a URL around the class locator
        try { classURL = new URL(classLoc); }
        catch (MalformedURLException e)
        {
            throw new ClassNotFoundException("Bad URL \"\" + classLoc +
                "\" given: " + e);
        }
        System.out.println("File = " + classURL.getFile());
        System.out.println("Host = " + classURL.getHost());
        // Get the file name from the URL
        String filename = classURL.getFile();
        // Make sure we're referencing a class file, then parse the class name
        if (! filename.endsWith(".class"))
            throw new ClassNotFoundException("Non-class URL given.");
        else
            className = filename.substring(0, filename.lastIndexOf(".class"));
        System.out.println("Classname = " + className);
        return className;
    }

    protected void initStream(String classLoc) throws IOException
    {
        // Ask the URL to open a stream to the class object
        classStream = classURL.openStream();
    }

    protected Class readClass(String classLoc, String className)
        throws IOException, ClassNotFoundException
    {
        // See how large the class file is...
        URLConnection conn = classURL.openConnection();
        int classSize = conn.getContentLength();
        System.out.println("Class file is " + classSize + " bytes.");

        // Read the class bytecodes from the stream
        DataInputStream dataIn = new DataInputStream(classStream);

        int avail = dataIn.available();
        System.out.println("Available = " + avail);
        System.out.println("URLClassLoader: Reading class from stream...");;
        byte[] classData = new byte[classSize];
        dataIn.readFully(classData);

        // Parse the class definition from the bytecodes
        Class c = null;
        System.out.println("URLClassLoader: Defining class...");;
        try { c = defineClass(classData, 0, classData.length); }
        catch (ClassFormatError e)
        {
            throw new ClassNotFoundException()
        }
    }
}
```

```

        "Format error found in class data.");
    }
    return c;
}
}

```

The parseClassName() implementation attempts to construct a URL object from the class locator. If an exception is raised, then an invalid URL has been passed in, and a ClassNotFoundException is thrown by the method. If the URL is successfully constructed, it is queried for the file name portion of the URL. The file suffix is checked to ensure that a ".class" file is being referenced, then the base of the file name is returned as the class name. The initStream() implementation simply calls openStream() on the URL object constructed from the class locator. If an IOException results, it is allowed to propagate up the call stack to loadClass(), which assumes that the class file addressed by the URL is inaccessible, and throws a ClassNotFoundException. Finally, the readClass() method reads the class bytecodes into a buffer by calling readFully() on the InputStream from the URL. An IOException will be allowed to propagate up to loadClass(), which throws a ClassNotFoundException. After successfully reading the bytecodes, readClass() next calls defineClass() to parse the class definition into a Class object, which is returned to the caller. If defineClass() generates a ClassFormatError, then a ClassNotFoundException is thrown, which loadClass() allows to propagate to the caller. Although catching an error, as opposed to an exception, goes against Java design doctrine, in this particular situation it may be a useful thing to do. Notice, however, that we've chosen to "convert" the error into a ClassNotFoundException. By doing this, we're saying that a format error in the loaded class should be considered as a missing class in the next level up the call stack.

We could implement other subclasses of the StreamClassLoader that use other network protocols to import Java bytecodes into the local runtime. We should note here that a Java-enabled browser uses something like our URLClassLoader to load classes for applets referenced in Web pages. A relative or absolute URL referring to a main applet class is passed to a network class loader, which does something along the lines of what happens in our readClass() method.

### 3.2.11 Network Programming with GUIs

Now that the basics of socket programming in Java have been covered, we can add some sophistication to our programs by providing them with graphical user interfaces (GUIs), which users have come to expect most software nowadays to provide. In order to concentrate upon the interface to each program, rather than upon the details of that program's processing, the examples used will simply provide access to some of the standard services, available via 'well known' ports.

#### Example

The following program uses the *Daytime* protocol to obtain the date and time from port 13 of user-specified host(s). It provides a text field for input of the host name by the user and a text area for output of the host's response. There are also two buttons, one that the user presses after entry of the host name and the other that closes down the program. The text area is 'wrapped' in a *JScrollPane*, to cater for long lines of output, while the buttons are laid out on a separate panel. The application frame itself will handle the processing of button presses, and so implements the *ActionListener* interface. The window-closing code (encapsulated in an anonymous *WindowAdapter* object) ensures that any socket that has been opened is closed before exit from the program.

```
import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.net.*;
import java.io.*;
import java.util.*;
public class GetRemoteTime extends JFrame implements ActionListener
{
    private JTextField hostInput;
    private JTextArea display;
    private JButton timeButton;
    private JButton exitButton;
    private JPanel buttonPanel;
    private static Socket socket = null;
    public static void main(String[] args)
    {
        GetRemoteTime frame = new GetRemoteTime();
        frame.setSize(400,300);
        frame.setVisible(true);
        frame.addWindowListener( new WindowAdapter()
        {
            public void windowClosing( WindowEvent event)
            {
                //Check whether a socket is open...
                if (socket != null)
                {
                    try
                    {
                        socket.close();
                    }
                    catch (IOException ioEx)
                    {
                        System.out.println("\nUnable to close link!\n");
                        System.exit(1);
                    }
                }
                System.exit(0);
            }
        });
    }
    public GetRemoteTime()
    {
        hostInput = new JTextField(20);
        add(hostInput, BorderLayout.NORTH);
        display = new JTextArea(10,15);
    }
}
```

```

//Following two lines ensure that word-wrapping occurs within the JTextArea
display.setWrapStyleWord(true);
display.setLineWrap(true);
add(new JScrollPane(display),
        BorderLayout.CENTER);
buttonPanel = new JPanel();
timeButton = new JButton("Get date and time ");
timeButton.addActionListener(this);
buttonPanel.add(timeButton);
exitButton = new JButton("Exit");
exitButton.addActionListener(this);
buttonPanel.add(exitButton);
add(buttonPanel,BorderLayout.SOUTH);
}

public void actionPerformed(ActionEvent event)
{
if (event.getSource() == exitButton) System.exit(0);
String theTime;
//Accept host name from the user...
String host = hostInput.getText();
final int DAYTIME_PORT = 13;
try
{
    //Create a Socket object to connect to specified host on relevant port...
    socket = new Socket(host, DAYTIME_PORT);
    //Create an input stream for the above Socket
    //and add string-reading functionality...
    Scanner input = new Scanner(socket.getInputStream());
    //Accept the host's response via the above stream...
    theTime = input.nextLine();
    //Add the host's response to the text in the JTextArea...
    display.append("The date/time at " + host + " is " + theTime + "\n");
    hostInput.setText("");
}
catch (UnknownHostException uhEx)
{
    display.append("No such host!\n");
    hostInput.setText("");
}
catch (IOException ioEx)
{
    display.append(ioEx.toString() + "\n");
}
finally
{
    try {
        if (socket!=null) socket.close(); //Close link to host...
    } catch(IOException ioEx) {
        System.out.println( "Unable to disconnect!"); System.exit(1); } } } }

```

Unfortunately, it is rather difficult nowadays to find a host that is running the *Daytime* protocol. Even if one does find such a host, it may be that the user's own firewall blocks the output from the remote server. If this is the case, then the user will be unaware of this until the connection times out which may take some time! The user is advised to terminate the program (with Ctrl-C) if the waiting time appears to be excessive. One possible way round this problem is to write one's own 'daytime server'...

To illustrate just how easy it is to provide a server that implements the *Daytime* protocol, example code for such a server is shown below. The program makes use of class *Date* from package *java.util* to create a *Date* object that will automatically hold the current day, date and time on the server's host machine. To output the date held in the *Date* object, we can simply use *println* on the object and its *toString* method will be executed implicitly (though we could specify *toString* explicitly, if we wished).

```

import java.net.*;
import java.io.*;
import java.util.Date;
public class DaytimeServer
{

```

```

public static void main(String[] args)
{
    ServerSocket server;
    final int DAYTIME_PORT = 13;
    Socket socket;
    try
    {
        server = new ServerSocket(DAYTIME_PORT);
        do
        {
            socket = server.accept();
            PrintWriter output = new PrintWriter( socket.getOutputStream(),true);
            Date date = new Date();
            output.println(date);
            //Method toString executed in line above.
            socket.close();
        }while (true);
    }
    catch (IOException ioEx)
    {
        System.out.println(ioEx); } } }

```

The server simply sends the date and time as a string and then closes the connection. If we run the client and server in separate command windows and enter *localhost* as our host name in the client's GUI, the result should look similar to that shown in Figure 2.7. Unfortunately, there is still a potential problem on some systems: since a low-numbered port (i.e., below 1024) is being used, the user may not have sufficient system rights to make use of the port. The solution in such circumstances is simple: change the port number (in both server and client) to a value above 1024. (E.g., change the value of *DAYTIME\_PORT* from 13 to 1300.)

Now for an example that checks a range of ports on a specified host and reports on those ports that are providing a service. This works by the program trying to create a socket on each port number in turn. If a socket is created successfully, then there is an open port; otherwise, an *IOException* is thrown (and ignored by the program, which simply provides an empty catch clause). The program creates a text field for acceptance of the required URL(s) and sets this to an initial default value. It also provides a text area for the program's output and buttons for checking the ports and for exiting the program.

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.net.*;
import java.io.*;

public class PortScanner extends JFrame implements ActionListener
{
    private JLabel prompt;
    private JTextField hostInput;
    private JTextArea report;
    private JButton seekButton, exitButton;
    private JPanel hostPanel, buttonPanel;
    private static Socket socket = null;
    public static void main(String[] args)
    {
        PortScanner frame = new PortScanner();
        frame.setSize(400,300);
        frame.setVisible(true);
        frame.addWindowListener(new WindowAdapter() {
            public void windowClosing( WindowEvent event)
            {
                //Check whether a socket is open...
                if (socket != null)
                {
                    try { socket.close(); }
                    catch (IOException ioEx) {
                        System.out.println( "\nUnable to close link!\n");

```

```

        System.exit(1);    }
    }
    System.exit(0);
} } );
}

public PortScanner() {
    hostPanel = new JPanel();
    prompt = new JLabel("Host name: ");
    hostInput = new JTextField("ivy.shu.ac.uk", 25);
    hostPanel.add(prompt);
    hostPanel.add(hostInput);
    add(hostPanel,BorderLayout.NORTH);
    report = new JTextArea(10,25);
    add(report,BorderLayout.CENTER);
    buttonPanel = new JPanel();
    seekButton = new JButton("Seek server ports ");
    seekButton.addActionListener(this);
    buttonPanel.add(seekButton);
    exitButton = new JButton("Exit");
    exitButton.addActionListener(this);
    buttonPanel.add(exitButton);
    add(buttonPanel,BorderLayout.SOUTH);
}

public void actionPerformed(ActionEvent event)
{
    if (event.getSource() == exitButton) System.exit(0);
    //Must have been the 'seek' button that was pressed, so clear the
    // output area of any previous output...
    report.setText("");
    //Retrieve the URL from the input text field...
    String host = hostInput.getText();
    try
    {
        //Convert the URL string into an InetAddress object...
        InetAddress theAddress = InetAddress.getByName(host);
        report.append("IP address: " + theAddress + "\n");
        for (int i = 0; i < 25; i++)
        {
            try {
                //Attempt to establish a socket on port i...
                socket = new Socket(host, i);
                //If no IOException thrown, it must be a service running on the port
                report.append( "There is a server on port " + i + ".\n");
                socket.close();
            }
            catch (IOException ioEx) {}// No server on this port
        }
    } catch (UnknownHostException uhEx) { report.setText("Unknown host!"); }
}
}

```

Unfortunately, remote users' firewalls may block output from most of the ports for this default server (or any other remote server), causing the program to wait for each of these port accesses to time out. This is likely to take a very long time indeed! The reader is strongly advised to use a local server for the testing of this program (and to get clearance from your system administrator for port scanning, to be on the safe side). Even when running the program with a suitable local server, be patient when waiting for output, since this may take a minute or so, depending upon your system.

### 3.2.12 Downloading Web Pages

Just one of the multitude of useful features of Java is its ability to render HTML pages as a browser would do, including the correct handling of hyperlinks contained within those pages. The class used to hold a Web page in a Java program is *JEditorPane*, which automatically renders HTML formatted text for any Web page that is downloaded via the *setPage* method of the *JEditorPane* object. (It also supports plain text and Rich Text Format, but attention will be devoted solely to HTML formatted text here.) The handling of hyperlinks requires only a modest amount of extra coding on the part of the Java programmer, as described in the following paragraphs.

If hyperlinks are contained within a downloaded page, a *HyperlinkEvent* is generated when the user clicks on one of these and must be handled by a *HyperlinkListener* (i.e., an object that implements the *HyperlinkListener* interface). A *HyperlinkEvent* is also generated when the user's mouse either moves over the hyperlink or moves away from it. Both of these actions may also cause processing activity to take place, if the application requires this (and may be ignored if it doesn't). In order to implement the *HyperlinkListener* interface, the listener object must provide a definition for method *hyperlinkUpdate*, which takes the *HyperlinkEvent* that occurred as its single argument. Method *hyperlinkUpdate*, of course, will specify the action that is to take place when a *HyperlinkEvent* occurs.

The first thing that method *hyperlinkUpdate* needs to ascertain is just which of the three possible *HyperlinkEvents* has just occurred. Class *HyperlinkEvent* contains a public inner class *EventType* that defines three constants for possible hyperlink event types:

- ACTIVATED* (user clicked a hyperlink);
- ENTERED* (mouse moved over a hyperlink);
- EXITED* (mouse moved away from a hyperlink).

Method *getEventType* (of class *HyperlinkEvent*) returns one of the above three constants.

#### Example

The following program displays the contents of a file at a user-specified URL, effectively acting as a simple browser. A text field is used to accept the user's URL string and a *JEditorPane* object is used to render the Web page at the specified URL. Since the *JEditorPane*'s size may very well be inadequate to display the full page, the pane is 'wrapped' in a *JScrollPane* object that will allow the user to scroll both vertically and horizontally on the page. The application frame states that it implements the *ActionListener* interface, thereby undertaking to provide a definition for the *actionPerformed* method. This method will specify the action to be carried out when the user presses <Enter> in the URL text field (both for the initial entry and for any subsequent, non-hyperlink changes of URL). Since this action is the same as that to be carried out when any hyperlink is clicked, this code has been placed inside a separate method called *showPage* (to avoid code duplication).

As for the *HyperlinkListener* interface, this is implemented by a private inner class called *LinkListener* (which means, of course, that it supplies a definition for method *hyperlinkUpdate*). If the *HyperlinkEvent* object indicates that the user clicked upon a hyperlink, then method *showPage* is called to display the page at the other end of the hyperlink. If, on the other hand, either of the other two possible hyperlink events occurred, then no action is taken in this application.

Method *showPage* renders the new page by calling method *setPage* of the *JEditorPane* object and then displays the URL of the page in the text field. Note that, if this latter step is not carried out, a runtime error will occur!

```
import java.awt.*;
import java.awt.event.*;
import java.net.*;
import java.io.*;
import javax.swing.*;
import javax.swing.event.*;
public class GetWebPage extends JFrame implements ActionListener {
private JLabel prompt; //Cues user to enter a URL.
private JTextField sourceName; //Holds URL string.
private JPanel requestPanel; //Contains prompt and URL string.
private JEditorPane contents; //Holds page.

public static void main(String[] args)
{
    GetWebPage frame = new GetWebPage();
    frame.setSize(700,500);
    frame.setVisible(true);
```

```

        frame.setDefaultCloseOperation(EXIT_ON_CLOSE);
    }
    public GetWebPage()
    {
        setTitle("Simple Browser");
        requestPanel = new JPanel();
        prompt = new JLabel("Required URL: ");
        sourceName = new JTextField(25);
        sourceName.addActionListener(this);
        requestPanel.add(prompt);
        requestPanel.add(sourceName);
        add(requestPanel, BorderLayout.NORTH);
        contents = new JEditorPane();
        //We don't want the user to be able to alter the
        //contents of the Web page display area, so...
        contents.setEditable(false);
        //Create object that implements HyperlinkListener interface...
        LinkListener linkHandler = new LinkListener();
        //Make the above object a HyperlinkListener for our JEditorPane object...
        contents.addHyperlinkListener(linkHandler);
        //'Wrap' the JEditorPane object inside a JScrollPane, to provide scroll bars
        add(new JScrollPane(contents), BorderLayout.CENTER);
    }
    public void actionPerformed(ActionEvent event)
    //Called when the user presses <Enter> after keying a URL into the text field
    //and also when a hyperlink is clicked.
    {
        showPage(sourceName.getText());
    }
    private class LinkListener implements HyperlinkListener
    {
        public void hyperlinkUpdate(HyperlinkEvent event)
        {
            if (event.getEventType() == HyperlinkEvent.EventType.ACTIVATED)
                showPage(event.getURL().toString());
            //Other hyperlink event types ignored.
        }
    }
    private void showPage(String location)
    {
        try
        {
            //Reset page displayed on JEditorPane...
            contents setPage(location);
            //Reset URL string in text field...
            sourceName.setText(location);
        }
        catch(IOException ioEx)
        {
            JOptionPane.showMessageDialog(this, "Unable to retrieve URL",
                                         "Invalid URL", JOptionPane.ERROR_MESSAGE);
        } } }

```

It is worth mentioning another class from the *java.net* package: the *URL* class. This class has six possible constructors, but the only one that is commonly used is the one that takes a *String* object as its single argument. Method *setPage* is also overloaded, allowing the user to specify the target page as either a *String* or an object of class *URL*. It was the first of these options that was used in the preceding example, since it would have been pointless to create a *URL* object from the string simply so that we could then pass the newly-created *URL* object to the other version of *setPage*. Indeed, it is often the case that we can use the URL string directly in Java, without having to create a *URL* object. However, creating such an object is occasionally unavoidable.

### 3.2.13 NIO – New I/O packages

This chapter introduces the main facilities of the “New I/O” packages. There are two important parts: the `java.nio.channels` package, which introduces the Selector and Channel abstractions, and the `java.nio` package, which introduces the Buffer abstraction.

#### 3.2.13.1 NIO advantages

Basic Java Sockets work well for small-scale systems. But when it comes to servers that have to deal with many thousands of clients simultaneously, certain issues arise. We saw signs of this in the thread-per-client approach is limited in terms of scalability because of the overhead associated with creating, maintaining, and swapping between threads. Using a thread *pool* saves on that kind of overhead while still allowing an implementor to take advantage of parallel hardware, but for protocols with long-lived connections, the size of the thread pool still limits the number of clients that can be handled simultaneously. Consider an instant messaging server that relays messages between clients. Clients must be continuously connected to receive messages, so the thread pool size limits the total number of clients that can be served. Increasing the thread pool size increases the thread-handling overhead without improving performance, because most of the time clients are idle.

If this were all there is to it, NIO might not be needed. Unfortunately, there are other, more subtle challenges involved with using threads for scalability. One is that the programmer has very little control over *which* threads receive service *when*. You can set a Thread instance’s *priority* (higher-priority threads get preference over lower-priority ones), but ultimately the priority is just “advice”—which thread is chosen to run next is entirely up to the implementation. Thus, if a programmer wants to ensure that certain connections get served before others, or impose a specific order of service, threads may make it harder to do that.

But the most important issue with threads is probably one we haven’t encountered yet. That’s because in our “echo service” examples, each client served is completely independent of all others; clients do not interact with each other or affect the state of the server. However, some (most) servers have some information—what we call “state”—that needs to be accessed or modified by different clients at the same time. Think of a service that allows citizen store serve parking spaces for one-hour blocks in a big city, for example. The schedule of who gets which space for which time blocks must be kept consistent; the server may also need to ensure that the same user does not reserve more than one space at a time. These constraints require that some state information (i.e., the schedule) be shared across all clients. This in turn requires that access to that state be carefully *synchronized* through the use of *locks* or other mutual exclusion mechanisms. Otherwise, since the scheduler can interleave program steps from different threads more or less arbitrarily, different threads that are trying to update the schedule concurrently might overwrite each other’s changes.

The need to synchronize access to shared state makes it significantly harder to think about both correctness and performance of a threaded server. The reasons for this added complexity are beyond the scope of this book, but suffice it to say that the use of the required synchronization mechanisms adds still more scheduling and context-switching overhead, over which the programmer has essentially no control.

Because of these complications, some programmers prefer to stick with a *single-threaded* approach, in which the server has only one thread, which deals with all clients—not sequentially, but all at once. Such a server cannot afford to block on an I/O operation with any one client, and must use *nonblocking I/O* exclusively. Recall that with nonblocking I/O, we specify the maximum amount of time that a call to an I/O method may block (including zero). We saw an example of this where we set a timeout on the accept operation (via the `setSoTimeout()` method of `ServerSocket`). When we call `accept()` on that `ServerSocket` instance, if a new connection is pending, `accept()` returns immediately; otherwise it blocks until either a connection comes in or the timer expires, whichever comes first. This allows a single thread to handle multiple connections. Unfortunately, the approach requires that we constantly *poll* all sources of I/O, and that kind of “busy waiting” approach again introduces a lot of overhead from cycling through connections just to find out that they have nothing to do.

What we need is a way to poll a *set of clients all at once*, to find out which ones need service. That is exactly the point of the Selector and Channel abstractions introduced in NIO. A Channel instance represents a “pollable” I/O target such as a socket (or a file, or a device). Channels can *register* an instance of class Selector. The `select()` method of Selector allows you to ask “Among the set of channels, which ones are currently ready to be serviced (i.e., accepted, read, or written)?” There are numerous details to be covered later, but that’s the basic motivation for Selector and Channel, both of which are part of the `java.nio.channels` package.

The other major feature introduced in NIO is the Buffer class. Just as selectors and channels give greater control and predictability of the overhead involved with handling many clients at once, Buffer enables more efficient,

predictable I/O than is possible with the Stream abstraction. The nice thing about the stream abstraction is that it hides the finiteness of the underlying buffering, providing the illusion of an arbitrary-length container. The bad thing is that implementing that illusion may require either lots of memory allocation or lots of context-switching, or both. As with threads, this overhead is buried in the implementation, and is therefore not controllable or predictable. That approach makes it easy to write programs, but harder to tune their performance. Unfortunately, if you use the Java Socket abstraction, streams are all you've got.

That's why channels are designed around the use of Buffer instances to pass data around. The Buffer abstraction represents a *finite-capacity* container for data—essentially, an array with associated pointers indicating where to put data in, and where to read data out. There are two main advantages to using Buffer. First, the overhead associated with reading from and writing to the buffer is exposed to the programmer. For example, if you want to put data into a buffer but there's not enough room, you have to do something to make more room (i.e., get some data out, or move data that's already there to make more room, or create a new instance). This represents extra work, but you (the programmer) control how, whether, and when it happens. A smart programmer, who knows the application requirements well, can often reduce overhead by tweaking these choices. Second, some specialized flavors of Buffermap operations on the Java object directly to operations on resources of the underlying platform (for example, to buffers in the operating system). This saves some copying of data between different address spaces—an expensive operation on modern architectures.

### 3.2.13.2 Using Channels with Buffers

As we said above, a Channel instance represents a connection to a device through which we can perform I/O. In fact the basic ideas are very similar to what we've already seen with plain sockets. For TCP, use the ServerSocketChannel and SocketChannel. There are other types of channels for other devices (e.g., FileChannel), and most of what we say here applies to them as well, although we do not mention them further. One difference between channels and sockets is that typically one obtains a channel instance by calling a static factory method:

```
SocketChannel clntChan = SocketChannel.open();
ServerSocketChannel servChan = ServerSocketChannel.open();
```

Channels do not use streams; instead, they send/receive data from/to buffers. An instance of Buffer or any of its subclasses can be viewed as a fixed-length sequence of elements of a single primitive Java type. Unlike streams, buffers have fixed, finite capacity, and internal (but accessible) state that keeps track of how much data has been put in or taken out; they behave something like queues with finite capacity. The Buffer class is abstract; you get a buffer by creating an instance of one of its subtypes, each of which is designed to hold one of the primitive Java types (with the exception of boolean). Thus each instance is a FloatBuffer, or an IntBuffer, or a ByteBuffer, etc. (The ByteBuffer is the most flexible of these and will be used in most of our examples.) As with channels, constructors are not typically used to create buffer instances; instead they are created either by calling allocate(), specifying a capacity:

```
ByteBuffer buffer = ByteBuffer.allocate(CAPACITY);
```

or by wrapping an existing array:

```
ByteBuffer buffer = ByteBuffer.wrap(byteArray);
```

Part of the power of NIO comes from the fact that channels can be made nonblocking. Recall that some socket operations can block indefinitely. For example, a call to accept() can block waiting for a client to connect; a call to read() can block until data arrives from the other end of a connection. In general, I/O calls that make/accept a connection or read/write data can block indefinitely until something happens in the underlying network implementations. A slow, lossy, or just plain broken network can cause an arbitrary delay. Unfortunately, in general we don't know if a method call will block before we make it. An important feature of the NIO channel abstraction is that we can make a channel nonblocking by configuring its blocking behavior:

```
clntChan.configureBlocking(false);
```

Calls to methods on a nonblocking channel always return immediately. The return value of such a call indicates the extent to which the requested operation was achieved. For example, a call to accept() on a nonblocking ServerSocketChannel returns the client SocketChannel if a connection is pending and null otherwise.

Let's construct a nonblocking TCP echo client. The I/O operations that may block include connecting, reading, and writing. With a nonblocking channel, these operations return immediately. We must repeatedly call these operations until we have successfully completed all I/O.

#### TCPEchoClientNonblocking.java

```
0 import java.net.InetSocketAddress;
```

```

1 import java.net.SocketException;
2 import java.nio.ByteBuffer;
3 import java.nio.channels.SocketChannel;
4
5 public class TCPEchoClientNonblocking {
6
7     public static void main(String args[]) throws Exception {
8
9         if ((args.length < 2) || (args.length > 3)) // Test for correct # of args
10            throw new IllegalArgumentException("Parameter(s): <Server> <Word> [<Port>]");
11
12        String server = args[0]; // Server name or IP address
13        // Convert input String to bytes using the default charset
14        byte[] argument = args[1].getBytes();
15
16        int servPort = (args.length == 3) ? Integer.parseInt(args[2]) : 7;
17
18        // Create channel and set to nonblocking
19        SocketChannel clntChan = SocketChannel.open();
20        clntChan.configureBlocking(false);
21
22        // Initiate connection to server and repeatedly poll until complete
23        if (!clntChan.connect(new InetSocketAddress(server, servPort))) {
24            while (!clntChan.finishConnect()) {
25                System.out.print(".");
26            }
27        }
28        ByteBuffer writeBuf = ByteBuffer.wrap(argument);
29        ByteBuffer readBuf = ByteBuffer.allocate(argument.length);
30        int totalBytesRcvd = 0; // Total bytes received so far
31        int bytesRcvd; // Bytes received in last read
32        while (totalBytesRcvd < argument.length) {
33            if (writeBuf.hasRemaining()) {
34                clntChan.write(writeBuf);
35            }
36            if ((bytesRcvd = clntChan.read(readBuf)) == -1) {
37                throw new SocketException("Connection closed prematurely");
38            }
39            totalBytesRcvd += bytesRcvd;
40            System.out.print(".");
41        }
42
43        System.out.println("Received: " + // convert to String per default charset
44                           new String(readBuf.array(), 0, totalBytesRcvd));
45        clntChan.close();
46    }
47 }

1      Get and convert arguments: lines 9–16
2      Create nonblocking SocketChannel: lines 19–20
3      Connect to server: lines 23–27 Because the socket is nonblocking, the call to connect() may return before the connection is established; the method returns true if the connection completes before it returns, false otherwise. In the latter case, any attempt to send/receive will throw a NotYetConnectedException, so we “poll” the status continually by calling finishConnect(), which returns false until the connection completes. The print operation demonstrates that we can perform other tasks while waiting for the connection to complete. Such a busy wait is generally wasteful; we do it here to illustrate the use of the methods.
4      Create read/write buffers: lines 28–29 We create the ByteBuffer instances we’ll use for writing and reading by wrapping the byte[] containing the string we want to send, and allocating a new instance the same size as that array, respectively.
5      Loop until we have sent and received all the bytes: lines 32–41 Call write() as long as the output buffer has anything left in it. The call to read() does not block but rather returns 0 when no data is available to return. Again, the print operation demonstrates that we can perform other tasks while waiting for the communication to complete.
6      Print the received data: lines 43–44
7      Close the channel: line 45 Like sockets, channels should be closed when they are no longer needed.

```

### 3.2.13.3 Selectors

The Selector class allows us to avoid the wasteful “busy waiting” approach we saw in the nonblocking client. Consider an Instant Messaging server, for example. Thousands of clients may be connected, but only a few (possibly none) have messages waiting to be read and relayed at any time. We need a way to block just until at

least one channel is ready for I/O, and to tell which channels are ready. NIO selectors do all of this. An instance of Selector can simultaneously check (and wait, if desired) for I/O opportunities on a set of channels. In technical terms, a selector is a multiplexor because a single selector can manage I/O on multiple channels.

To use a selector, create it (using the static factory method open()) and *register* it with the channels that you wish to monitor (note that this is done via a method of the *channel*, not the selector). Finally, call the selector's select() method, which blocks until one or more channels are ready for I/O or a timeout expires. When select() returns, it tells you the number of channels ready for I/O. Now, in a single thread, we can check for ready I/O on several channels by calling select(). If no I/O becomes ready after a certain amount of time, select() returns 0 and allows us to continue on with other tasks.

Let's look at an example. Suppose we want to implement an echo server using channels and a selector without using multiple threads or busy waiting. To make it easier to use this basic server pattern with different protocols, we have factored out the protocol-specific details of how each type of I/O (accepting, reading, and writing) is handled through the channel. TCP Protocol defines the interface between the generic TCPSelectorServer.java and the specific protocol. It includes three methods, one for each form of I/O; the server simply invokes the appropriate method once a channel becomes ready.

### **TCPProtocol.java**

```
0 import java.nio.channels.SelectionKey;
1 import java.io.IOException;
2
3 public interface TCPProtocol {
4     void handleAccept(SelectionKey key) throws IOException;
5     void handleRead(SelectionKey key) throws IOException;
6     void handleWrite(SelectionKey key) throws IOException;
7 }
```

Now for the server. We create a selector and register it with a ServerSocketChannel for each socket on which the server listens for incoming client connections. Then we loop forever, invoking select(), and calling the appropriate handler routine for whatever type of I/O is appropriate.

### **TCPServerSelector.java**

```
0 import java.io.IOException;
1 import java.net.InetSocketAddress;
2 import java.nio.channels.SelectionKey;
3 import java.nio.channels.Selector;
4 import java.nio.channels.ServerSocketChannel;
5 import java.util.Iterator;
6
7 public class TCPServerSelector {
8
9     private static final int BUFSIZE = 256; // Buffer size (bytes)
10    private static final int TIMEOUT = 3000; // Wait timeout (milliseconds)
11
12    public static void main(String[] args) throws IOException {
13
14        if (args.length < 1) { // Test for correct # of args
15            throw new IllegalArgumentException("Parameter(s): <Port> ...");
16        }
17
18        // Create a selector to multiplex listening sockets and connections
19        Selector selector = Selector.open();
20
21        // Create listening socket channel for each port and register selector
22        for (String arg : args) {
23            ServerSocketChannel listnChannel = ServerSocketChannel.open();
24            listnChannel.socket().bind(new InetSocketAddress(Integer.parseInt(arg)));
25            listnChannel.configureBlocking(false); // must be nonblocking to register
26            // Register selector with channel. The returned key is ignored
27            listnChannel.register(selector, SelectionKey.OP_ACCEPT);
28        }
29
30        // Create a handler that will implement the protocol
31        TCPProtocol protocol = new EchoSelectorProtocol(BUFSIZE);
```

```

32
33     while (true) { // Run forever, processing available I/O operations
34         // Wait for some channel to be ready (or timeout)
35         if (selector.select(TIMEOUT) == 0) { // returns # of ready chans
36             System.out.print(".");
37             continue;
38         }
39
40         // Get iterator on set of keys with I/O to process
41         Iterator<SelectionKey> keyIter = selector.selectedKeys().iterator();
42         while (keyIter.hasNext()) {
43             SelectionKey key = keyIter.next(); // Key is bit mask
44             // Server socket channel has pending connection requests?
45             if (key.isAcceptable()) {
46                 protocol.handleAccept(key);
47             }
48             // Client socket channel has pending data?
49             if (key.isReadable()) {
50                 protocol.handleRead(key);
51             }
52             // Client socket channel is available for writing and
53             // key is valid (i.e., channel not closed)?
54             if (key.isValid() && key.isWritable()) {
55                 protocol.handleWrite(key);
56             }
57             keyIter.remove(); // remove from set of selected keys
58         }
59     }
60 }
61 }
```

1       Setup: lines 14–19 Verify at least one argument, create a Selector instance.

2.       Create a ServerSocketChannel for each port: lines 22–28

- Create a ServerSocketChannel: line 23

- Make it listen on the given port: line 24 We have to fetch the underlying ServerSocket and invoke its bind() method on the port given as argument. Any argument other than a number in the appropriate range will result in an IOException.

- Make it nonblocking: line 25 Only nonblocking channels can register selectors, so we configure the blocking state appropriately.

- Register selector with channel: line 27 We indicate our interest in the “accept” operation during registration.

3       Create protocol handler: line 31 To get access to the handler methods for the Echo protocol, we create an instance of the EchoSelectorProtocol, which exports the required methods.

4       Loop forever, waiting for I/O, invoking handler: lines 33–59

- Select: line 35 This version of the select() method blocks until some channel becomes ready or until the timeout expires. It returns the number of ready channels; zero indicates that the timeout expired, in which case we print a dot to mark the passage of time and iterate.

- Get selected key set: line 41 The selectedKeys() method returns a Set, for which we get an Iterator. The set contains the SelectionKey (created at registration time) of each channel that is ready for one of the I/O operations of interest (specified at registration time).

- Iterate over keys, checking ready operations: lines 42–58 For each key, we check whether it is ready for accept(), readable, and/or writable, invoking the appropriate handler method to perform the indicated operation in each case.

- Remove the key from the set: line 57 The select() operation only *adds* to the set of selected keys associated with a Selector. Therefore if we do not remove each key as we process it, it will *remain* in the set across the next call to select(), and a useless operation may be invoked on it.

TCPServerSelector is protocol agnostic for the most part; only the single line of code assigning the value of protocol is protocol-specific. All protocol details are contained in the implementation of the TCPProtocol interface. EchoSelectorProtocol provides an implementation of the handlers for the Echo protocol. You could easily write your own protocol handlers for other protocols or performance improvements on our Echo protocol handler implementation.

### **EchoSelectorProtocol.java**

```

0 import java.nio.channels.SelectionKey;
1 import java.nio.channels.SocketChannel;
2 import java.nio.channels.ServerSocketChannel;
3 import java.nio.ByteBuffer;
4 import java.io.IOException;
5
6 public class EchoSelectorProtocol implements TCPProtocol {
7
8     private int bufSize; // Size of I/O buffer
9 }
```

```

10 public EchoSelectorProtocol(int bufSize) {
11     this.bufSize = bufSize;
12 }
13
14 public void handleAccept(SelectionKey key) throws IOException {
15     SocketChannel clntChan = ((ServerSocketChannel) key.channel()).accept();
16     clntChan.configureBlocking(false); // Must be nonblocking to register
17     // Register the selector with new channel for read and attach byte buffer
18     clntChan.register(key.selector(), SelectionKey.OP_READ,
19                         ByteBuffer.allocate(bufSize));
20 }
21
22 public void handleRead(SelectionKey key) throws IOException {
23     // Client socket channel has pending data
24     SocketChannel clntChan = (SocketChannel) key.channel();
25     ByteBuffer buf = (ByteBuffer) key.attachment();
26     long bytesRead = clntChan.read(buf);
27     if (bytesRead == -1) { // Did the other end close?
28         clntChan.close();
29     } else if (bytesRead > 0) {
30         // Indicate via key that reading/writing are both of interest now.
31         key.interestOps(SelectionKey.OP_READ | SelectionKey.OP_WRITE);
32     }
33 }
34
35 public void handleWrite(SelectionKey key) throws IOException {
36     /*
37      * Channel is available for writing, and key is valid (i.e., client channel
38      * not closed).
39      */
40     // Retrieve data read earlier
41     ByteBuffer buf = (ByteBuffer) key.attachment();
42     buf.flip(); // Prepare buffer for writing
43     SocketChannel clntChan = (SocketChannel) key.channel();
44     clntChan.write(buf);
45     if (!buf.hasRemaining()) { // Buffer completely written?
46         // Nothing left, so no longer interested in writes
47         key.interestOps(SelectionKey.OP_READ);
48     }
49     buf.compact(); // Make room for more data to be read in
50 }
51 }

```

- 1 Declaration of implementation of the TCPProtocol interface: line 6
- 2 Member variables and constructor: lines 8–12 Each instance contains the size of buffer to be created for each client channel.
- 3 handleAccept(): lines 14–20
  - Get channel from key and accept connection: line 15 The channel() method returns the Channel that created the key at registration time. (We know it's a ServerSocketChannel because that's the only kind we registered with that supports the “accept” operation.) The accept() method returns a SocketChannel for the incoming connection.
  - Make nonblocking: line 16. Again, we cannot register with a blocking channel.
  - Register selector with channel: lines 18–19 As with the channel, we can retrieve the Selector associated with the SelectionKey via its selector() method. We create a new ByteBuffer of the required size, and pass it as argument to register(). It will be associated as an attachment to the SelectionKey instance returned by the register() method. (We ignore the returned key now, but will access it through the selected keys set if the channel becomes ready for I/O.)
4. handleRead(): lines 22–33
  - Get channel associated with key: line 24 We know this is a SocketChannel because it supports reading.
  - Get buffer associated with key: line 25 When the connection was associated, a ByteBuffer was attached to this SelectionKey instance.
  - Read from the channel: line 27
  - Check for end of stream and close channel: lines 27–28 If the read() returns `-1`, we know the underlying connection closed, and close the channel in that case. Closing the channel removes its associated key from the selector's various sets.
  - If data received, indicate interest in writing: lines 29–31 Note that we are still interested in reading, although there may not be any room left in the buffer.
5. handleWrite(): lines 35–50
  - Retrieve buffer containing received data: line 41 The ByteBuffer attached to the given SelectionKey contains data read earlier from the channel.
  - Prepare buffer for writing: line 42 The Buffer's internal state indicates where to *put data next*, and how much *space* is left. The flip() operation modifies the state so it indicates from where to *get data* for the write() operation, and how much *data* is left. (This is explained in detail in the next section.) The effect is that the write operation will start consuming the data produced by the earlier read.

- Get channel: line 43
  - Write to channel: line 44
- If buffer empty, lose interest in writing: lines 45–48 If there is no received data left in the buffer, we modify the interest set associated with the key so that it indicates only read is of interest.
- Compact the buffer: line 49 If there is data remaining in the buffer, this operation moves it to the front of the buffer so more data can be read on the next iteration. In any case, the operation resets the state so the buffer is again ready for reading. Note that the buffer associated with a channel is always set up for reading except when control is inside the handleWrite() method.

We are now ready to delve into the details of the three main NIO abstractions.

### 3.2.13.4 Buffers in Detail

As you've already seen, in NIO data is read into and written from buffers. Channels read data into buffers. We then access the data through the buffer. To write data, we first fill the buffer with data in the order we wish to send it. Basically, a buffer is just a list where all of the elements are a single primitive type (typically bytes). A buffer is fixed-length; it cannot expand like some other classes (e.g., List, StringBuffer, etc). Note that ByteBuffer is commonly used because 1) it provides methods for reading and writing other types, and 2) the channel read/write methods accept only ByteBuffers.

#### Buffer Indices

A buffer goes beyond just storing a list of elements. It has internal state that keeps track of the current position when reading data from or writing data to the buffer, as well as the end of valid data for reading, etc. To do this, each buffer maintains four indices into its list of elements; they are shown in Table 3.3.

**Table 3.3: Buffer Internal State**

Index	Description	Accessor/Mutator/Usage
<i>capacity</i>	Number of elements in buffer (Immutable)	int capacity()
<i>position</i>	Next element to read/write	int position()
	(numbered from 0)	Bufferposition(int newPosition)
<i>limit</i>	First unreadable/unwritable element	int limit()
		Bufferlimit(int newLimit)
<i>mark</i>	User-chosen prev. value of <i>position</i> , or 0	Buffer mark()
		Buffer reset()

The distance between the *position* and *limit* tells us the number of bytes available for getting/putting. Java provides two convenience methods for evaluating this distance.

#### ByteBuffer: Remaining Bytes

```
boolean hasRemaining()
int remaining()
```

`hasRemaining()` returns true if at least one element is available, and `remaining()` returns the number of elements available.

The following relationships among these variables are maintained as an invariant:

$$0 \leq \text{mark} \leq \text{position} \leq \text{limit} \leq \text{capacity}$$

The *mark* value “remembers” a position so you can come back to it later; the `reset()` method returns the *position* to the value it had when `mark()` was last called (unless doing so would violate the above invariant).

#### Buffer Creation

Typically, we create buffers either by allocation or by wrapping an array of primitives. The static factory methods for creating a ByteBuffer are shown in Table 3.4, along with the initial values of *capacity*, *position*, and *limit* for the returned instance. The initial value of *mark* is undefined for all newly created Buffer instances; attempts to `reset()` the *position* before calling `mark()` result in an `InvalidMarkException`.

To allocate a fresh instance, we simply call the static `allocate()` method for the type of buffer we want, specifying the number of elements:

```
ByteBuffer byteBuf = ByteBuffer.allocate(20);
DoubleBuffer dblBuf = DoubleBuffer.allocate(5);
```

**Table 3.4: ByteBuffer Creation Methods**

Method	Capacity	Position	Limit
ByteBufferallocate(int capacity)	capacity	0	capacity
ByteBufferallocateDirect(int capacity)	capacity	0	capacity
ByteBufferwrap(byte[] array)	array.length	0	array.length
ByteBufferwrap(byte[] array, int offset, int length)	array.length	offset	offset + length

Here byteBuf holds 20 bytes, and dblBuf holds 5 Java doubles. These buffers are fixed-size so they can never be expanded or contracted. If you find that the buffer you just allocated is too short, your only option is to allocate a new, correctly sized buffer.

We can also create a buffer from an existing array by calling the static wrap() method and passing the array to be wrapped

```
byteArray[] = new byte[BUFFERSIZE];
// ...Fill array...
ByteBuffer byteWrap = ByteBuffer.wrap(byteArray);
ByteBuffer subByteWrap = ByteBuffer.wrap(byteArray, 3, 3);
```

A buffer created by wrapping contains the data from the wrapped array. In fact, wrap() simply creates a buffer with a reference to the wrapped array, called the backing array. Any change to the data in the backing array changes the data in the buffer and vice versa. If we specify an offset and length to wrap(), the buffer is backed by the entire array with position and limit initially set to offset and offset + length. The elements preceding the offset and following the length are still accessible via the buffer.

Creation of a buffer by allocation isn't really so different from wrapping. The only real difference is that allocate() creates its own backing array. You can get a reference to this backing array by calling array() on the buffer. You can even get the offset into the backing array of the first element used by the buffer by calling arrayOffset(). A buffer created with wrap() with a nonzero offset still has an array offset of 0.

So far, all of our buffers store data in Java-allocated backing arrays. Typically, the underlying platform (operating system) cannot use these buffers to perform I/O. Instead the OS must use its own buffers for I/O and copy the results to/from the buffer's backing array. Such copying can get expensive, especially if there are many reads and writes requiring copy-ing. Java NIO provides *direct buffers* as a way around this problem. With a direct buffer, Java allocates the backing store of the buffer from storage that the platform can use for I/O directly, so copying is unnecessary. Such low-level, native I/O generally operates at the byte level, so only ByteBuffers can be directly allocated.

```
ByteBuffer byteBufDirect = ByteBuffer.allocateDirect(BUFFERSIZE);
```

You can test whether a buffer is direct by calling isDirect(). Since a direct buffer does not have a backing array, calling array() or arrayOffset() on a direct buffer will throw an UnsupportedOperationException. There are a few caveats to remember when considering whether to use direct buffers. Calling allocateDirect() doesn't guarantee you are allocated a direct buffer—your platform or JVM may not support this operation, so you have to call isDirect() after attempting to allocate. Also, allocation and deallocation of a direct buffer is typically more expensive than for nondirect buffers, because the backing store of a direct buffer typically lives outside the JVM, requiring interaction with the operating system for management. Consequently, you should only allocate direct buffers when they will be used for a long time, over many I/O operations. In fact, it is a good idea to use direct buffers only if they provide a measurable increase in performance over nondirect buffers.

### **Storing and Retrieving Data**

Once you have a buffer, it's time to use it to hold data. As “containers” for data, buffers are used for both input and output; this is different from streams, which transfer data in only one direction. We place data into a buffer using put(), and retrieve data from a buffer using get(). A channel read() implicitly calls put(), and a channel write() implicitly calls get() on the given buffer. Below we present the get() and put() methods for ByteBuffer; however, the other buffer types have similar methods.

*ByteBuffer: Getting and putting bytes*

Relative:

```
byte get()
ByteBuffer get(byte[] dst)
ByteBuffer get(byte[] dst, int offset, int length)
ByteBuffer put(byte b)
ByteBuffer put(byte[] src)
```

```
ByteBuffer put(byte[] src, int offset, int length)
ByteBuffer put(ByteBuffer src)
```

Absolute:

```
byte get(int index)
ByteBuffer put(int index, byte b)
```

There are two types of get() and put(): relative and absolute. The relative variants get/put data from/to the “next” location in the buffer according to the value of *position*, and then increment *position* by an appropriate amount (that is, by one for the single-byte form, by array.length for the array form, and by length for the array/offset/length form). Thus, each call to put() appends after elements already contained in the buffer, and each call to get() retrieves the next element from the buffer. However, if doing so would cause *position* to go past *limit*, a get() throws a BufferUnderflowException, while a put() throws a BufferOverflowException. For example, if the destination array passed to get() is longer than the available remaining elements in the buffer, get() throws BufferUnderflowException; partial gets/puts are not allowed. The absolute variants of get() and put() take a specific index for getting and putting data; *the absolute forms do not modify position*. They do, however, throw IndexOutOfBoundsException if the given index exceeds *limit*.

The class ByteBuffer provides additional methods for relative and absolute get/put of other types besides bytes; in this way, it’s like a DataOutputStream.

#### *ByteBuffer: Getting and putting Java multibyte primitives*

```
_type_ get_Type_()
$type_ get_Type_(int index)
ByteBuffer put_Type_(_type_ value)
ByteBuffer put_Type_(int index, _type_ value)
where “_Type_” stands for one of Char, Double, Int, Long, Short
and “_type_” stands for one of char, double, int, long, short
```

Each call to a relative put() or get() advances the value of *position* by the length of the particular parameter type: 2 for short, 4 for int, etc. However, if doing so would cause *position* to exceed *limit*, a BufferUnderflowException (get) or BufferOverflowException (put) is thrown: partial gets and puts are not allowed. In the case of under/overflow, *position* does not change.

You may have noticed that many get/put methods return a ByteBuffer. In fact, they return the same instance of ByteBuffer that was passed as an argument. This allows *call chaining*, where the result of the first call is used to make a subsequent call. For example, we can put the integers 1 and 2 in the ByteBuffer instance *myBuffer* as follows:

```
myBuffer.putInt(1).putInt(2);
```

Multibyte values have a byte order, namely big-or little-endian. By default Java uses big-endian. You can get and set the order in which multibyte values are written to a byte buffer, using the built-in instances ByteOrder.BIG\_ENDIAN and ByteOrder.LITTLE\_ENDIAN.

#### *ByteBuffer: Byte ordering in buffer*

```
ByteOrder order()
ByteBuffer order(ByteOrder order)
```

The first method returns the buffer’s current byte order, as one of the constants of the ByteOrder class. The second allows you to set the byte order used to write multibyte quantities.

Let’s look at an example using byte order:

```
ByteBuffer buffer = ByteBuffer.allocate(4);
buffer.putShort((short) 1);
buffer.order(ByteOrder.LITTLE_ENDIAN);
buffer.putShort((short) 1);
// Predict the byte values for buffer and test your prediction
```

With all of this talk about byte ordering, you may be wondering about the byte order of your processor. ByteOrder defines a method to answer your question:

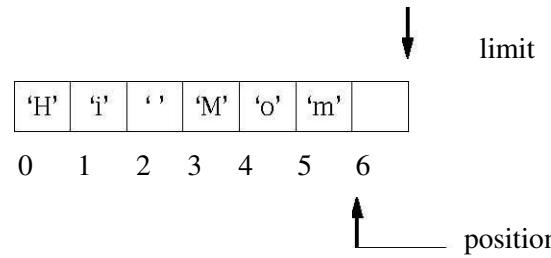
#### *ByteOrder: Finding byte order*

```
static final ByteOrder BIG_ENDIAN
static final ByteOrder LITTLE_ENDIAN
static ByteOrder nativeOrder()
```

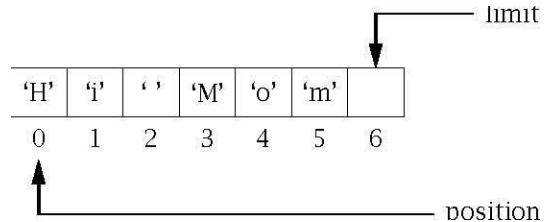
The method nativeOrder() returns one of the two constants BIG\_ENDIAN or LITTLE\_ENDIAN.

#### *PreparingBuffers: clear(), flip(), and rewind()*

Before using a buffer for input or output, we need to make sure the buffer is correctly prepared with *position* and *limit* set to the proper values. Consider a CharBuffer created with capacity seven, which has been populated by successive calls to put() or read():

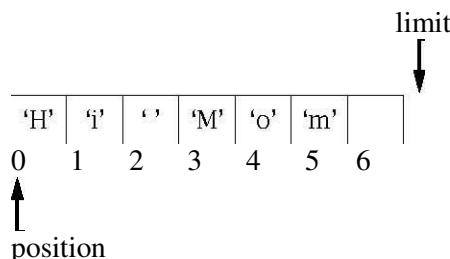


If we now want to use this buffer to do a channel write, since `write()` will start getting data at *position*, and stop at *limit*, we need to set *limit* to the current value of *position*, and set *position* to 0.



We could handle this ourselves, but fortunately Java provides some convenience methods to do the work for us; they are shown in Table 3.5.

Note that these methods *do not change the buffer's data*, only its indices. The `clear()` method prepares the buffer to accept new data from a buffer put or channel read by setting *position* to zero and *limit* to *capacity*. Continuing the example above, after `clear()` the situation looks like this:



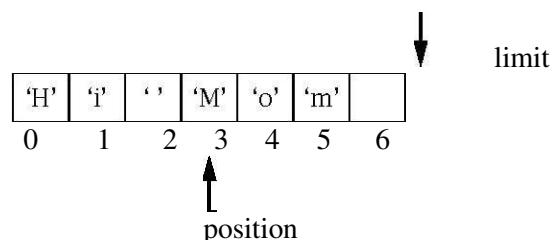
**Table 3.5. Instance Methods of ByteBuffer**

Resulting Value of				
ByteBuffer Method	Prepares Buffer for	Position	Limit	Mark
<code>clear()</code>	<code>read()/put()</code> into buffer	0	<i>capacity</i>	undefined
<code>flip()</code>	<code>write()/get()</code> from buffer	0	<i>position</i>	undefined
<code>rewind()</code>	<code>rewrite()/get()</code> from buffer	0	unchanged	undefined

Subsequent calls to `put()/read()` fill the buffer, starting from the first element and filling up to the limit, which is set to the capacity.

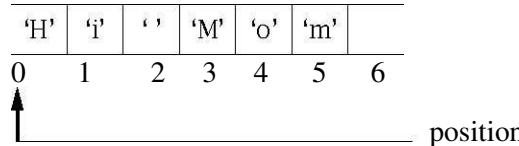
```
// Start with buffer in unknown state
buffer.clear(); // Prepare buffer for input, ignoring existing state
channel.read(buffer); // Read new data into buffer, starting at first element
```

Despite its name, `clear()` doesn't actually change the buffer's data; it simply resets the buffer's main index values. Consider a buffer recently populated with data (say, 3 characters) from `put()` and/or `read()`. The *position* value indicates the first element that does not contain valid data:



The `flip()` method prepares for data transfer *out* of the buffer, by setting *limit* to the current position and *position* to zero:





Subsequent calls to `get()`/`write()` retrieve data from the buffer starting from the first element and proceeding up to the *limit*. Here is an example of `flip()`'s usage:

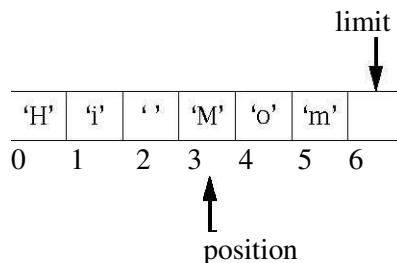
```
// ... put data in buffer with put() or read() ...
buffer.flip(); // Set position to 0, limit to old position
while (buffer.hasRemaining()) // Write buffer data from the first element up to lim
channel.write(buffer);
```

Suppose you've written some or all of the data from a buffer and you'd like to go back to the beginning of the buffer to write the same information again (for example, you want to send it on another channel). The `rewind()` method sets *position* to zero and invalidates the *mark*. It's similar to `flip()` except *limit* remains unchanged. When might you use this? Well, you might want to write everything you send over the network to a logger:

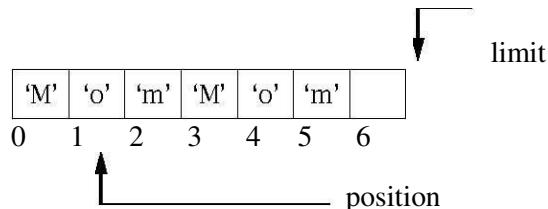
```
// Start with buffer ready for writing
while (buffer.hasRemaining()) // Write all data to network
networkChannel.write(buffer);
buffer.rewind(); // Reset buffer to write again
while (buffer.hasRemaining()) // Write all data to logger
loggerChannel.write(buffer);
```

### Compacting Data in a Buffer

The `compact()` operation copies the elements between *position* and *limit* to the start of the buffer, to make room for subsequent `put()`/`read()` calls. The value of *position* is set to the length of the copied data, the value of *limit* is set to the capacity, and *mark* becomes undefined. Consider the following buffer state before `compact()` is called:



Here is the situation after `compact()`:



Why use this operation? Suppose you have a buffer for writing data. Recall that a nonblocking call to `write()` only uses the data it can send without blocking; therefore, `write()` will not necessarily send all elements of the buffer. Now you need to `read()` new data into the buffer, after the unwritten data. One way to handle this is to simply set `position=limit` and `limit=capacity`. Of course, you'll need to reset these values later, after reading but before you call `write()` again. The problem is that eventually the buffer will run out of space; in the figures above, there would only be room for one more byte. Moreover, any space at the beginning of the buffer is wasted. This is exactly the problem `compact()` is designed to solve. By calling `compact()` after the `write()` but before the `read()` that will add more data, we move all the “left over” data to the start of the buffer, freeing up the maximum space for new data.

```
// Start with buffer ready for reading
while (channel.read(buffer) != -1) {
buffer.flip();
channel.write(buffer);
buffer.compact();
}
while (buffer.hasRemaining())
channel.write(buffer);
```

Note, however, that as we mentioned at the beginning of the chapter, copying data is a rather expensive operation, so compact() should be used sparingly.

### **BufferPerspectives: *duplicate()*, *slice()*, etc.**

NIO provides several ways of creating a new buffer that shares content with a given buffer, but differs on the processing of the elements. Basically, the new buffer has its own independent state variables (position, limit, capacity, and mark) but shares the backing storage with the original buffer. Any changes to the new buffer are shared with the original. Think of this as an alternate perspective on the same data. Table 5.4 lists the relevant methods.

The *duplicate()* method creates a new buffer that shares the content of the original buffer. The new buffer's position, limit, mark, and capacity initially match the original buffer's index values; however, the values are independent. Since the content is shared, changes to the original buffer or any duplicates will be visible to all. Let's return to our example above where you want to write everything you send over the network to a logger.

```
// Start with buffer ready for writing
ByteBuffer logBuffer = buffer.duplicate();
while (buffer.hasRemaining()) // Write all data to network
    networkChannel.write(buffer);
while (logBuffer.hasRemaining()) // Write all data to logger
    loggerChannel.write(logBuffer);
```

Note that with buffer duplication, writing to the network and log could be done in parallel using different threads.

The *slice()* method creates a new buffer that shares some subsequence of the original buffer. The new buffer's position is zero, and its limit and capacity are both equal to the difference between the limit and position of the original buffer. *slice()* sets the new buffer's array offset to the original buffer's position; however, calling *array()* on the new buffer still returns the entire array.

Channel reads and writes take only ByteBuffers; however, we may be interested in communicating using other primitive types. A ByteBuffer can create a separate "view buffer" that interprets its contents as some other primitive type (e.g., CharBuffer). Data of the new type can then be read from (and written to, although that is an optional operation) this buffer. The new buffer shares the backing storage of the original ByteBuffer; therefore, changes to either buffer are seen in both new and original buffers. A newly created view buffer has its *position* set to zero, and its contents start at the original buffer's *position*. This is very similar to the *slice()* operation; however, since the view buffer operates over multibyte elements, the capacity and limit of the new buffer is the remaining number of bytes divided by the number of bytes in the corresponding primitive type (e.g., divide by 8 when creating a DoubleBuffer).

Let's look at an example. Suppose you have received (via some Channel) a message that consists of a single byte, followed by a number of two-byte integers (i.e., shorts), in big-endian order. Because the message arrives over a Channel, it's in a ByteBuffer, buf. The first byte of the message contains the number of two-byte integers that make up the rest of the message. You might call *buf.getShort()* the number of times indicated by the first byte. Or you can get all the integers at once, like this:

```
// ...get message by calling channel.read(buf) ...
int numShorts = (int)buf.get();
if (numShorts < 0) {
    throw new SomeException();
} else {
    short[] shortArray = new short[numShorts];
    ShortBuffer sbuf = buf.asShortBuffer();
    sbuf.get(shortArray); // note: will throw if header was incorrect!
}
```

The *asReadOnlyBuffer()* method works just like *duplicate()* except that all mutator methods on the new buffer will always throw a *ReadOnlyBufferException*. This includes all forms of *put()*, *compact()*, etc. Even calls to *array()* and *arrayOffset()* for a non-direct buffer throw this exception. Of course, changes to the non-read-only buffer that generated this read-only buffer will still be shared. Like a buffer created with *duplicate()*, read-only buffers have independent buffer state variables. You can use the *isReadOnly()* method to test if a buffer is read-only. If a buffer is already read-only, calling *duplicate()* or *slice()* will create a read-only buffer.

### **Character Coding**

Characters are encoded as sequences of bytes, and that there are various mappings (called charsets) between sets of characters and byte sequences. Another use of NIO buffers is to convert among various charsets. To use this

facility, you need to know about two additional classes in the `java.nio.charset` package: `CharsetEncoder` and `CharsetDecoder`.

To encode, use a `Charset` instance to create an encoder and call `encode`:

```
Charset charSet = Charset.forName("US-ASCII");
CharsetEncoder encoder = charSet.newEncoder();
ByteBuffer buffer = encoder.encode(CharBuffer.wrap("Hi mom"));
```

To decode, use the `Charset` instance to create a decoder and call `decode`:

```
CharsetDecoder decoder = charSet.newDecoder();
CharBuffer cBuf = decoder.decode(buffer);
```

While this approach certainly works, it can be inefficient when coding multiple times. For example, each call to `encode/decode` creates a new `Byte/CharBuffer`. Other inefficiencies crop up relating to coder creation and operation.

```
encoder.reset();
if (encoder.encode(CharBuffer.wrap("Hi "),buffer,false) ==
CoderResult.OVERFLOW) {
// ... deal with lack of space in buffer ...
}
if (encoder.encode(CharBuffer.wrap("Mom"),buffer,true) ==
CoderResult.OVERFLOW) {
// ... ditto ...
}
encoder.flush(buffer);
```

The `encode()` method converts the given `CharBuffer` into a byte sequence and writes the bytes to the given buffer. If the buffer is too small, `encode()` returns a `CoderResult.OVERFLOW`. If the input is completely consumed and the encoder is ready for more, `CoderResult.UNDERFLOW` is returned; otherwise the input is malformed in some way, and a `CoderResult` object is returned that indicates the nature and location of the problem. We set the final boolean parameter to true only when we have reached the end of input to the encoder. `flush()` pushes any buffered encoder state to the buffer. Note that it is not strictly necessary to call `reset()`, which sets up the encoder's internal state so it can encode again, on a freshly created encoder.

### Stream (TCP) Channels in Detail

Stream channels come in two varieties: `SocketChannel` and `ServerSocketChannel`. Like its `Socket` counterpart, a `SocketChannel` is a communication channel for connected endpoints.

#### *SocketChannel: Creating, connecting, and closing*

```
static SocketChannel open(SocketAddress remote)
static SocketChannel open()
boolean connect(SocketAddress remote)
boolean isConnected()
void close()
boolean isOpen()
Socket socket()
```

A `SocketChannel` is created by calling the `open()` factory method. The first form of `open()` takes a `SocketAddress` and returns a `SocketChannel` connected to the specified server; note that this method may block for an indefinite period. The parameter-less form of `open()` creates an unconnected `SocketChannel`, which may be connected to an endpoint with the `connect()` method. When you are finished with a `SocketChannel`, call the `close()` method. One important point is that each instance of `SocketChannel` “wraps” a basic Java `Socket`, which you may access using the `socket()` method. This will allow you to call basic `Socket` methods to bind, set socket options, etc. After you create and connect your `SocketChannel`, you perform I/O with the channel's read and write methods.

#### *SocketChannel: Reading and writing*

```
int read(ByteBuffer dst)
long read(ByteBuffer[] dsts)
long read(ByteBuffer[] dsts, int offset, int length)
int write(ByteBuffer src)
long write(ByteBuffer[] srcts)
long write(ByteBuffer[] srcts, int offset, int length)
```

The most basic form of `read` takes a single `ByteBuffer` and reads up to the number of bytes remaining in the buffer. The other form of `read` takes an array of `ByteBuffers` and reads up to the number of bytes remaining in all of the buffers by filling each buffer in array order. This is called a *scattering read* because it scatters the incoming bytes over multiple buffers. It's important to note that the scattering read isn't obligated to fill all the buffer(s); the total amount of buffer space is simply an upper bound. The most basic form of `write` takes a single `ByteBuffer` and attempts to write the bytes remaining in the buffer to the channel. The other form of `write` takes an array of `ByteBuffers` and

attempts to write the bytes remaining in all buffers. This is called a *gathering write* because it gathers up bytes from multiple buffers to send together. Like its ServerSocketcounter part, a ServerSocketChannel is a channel for listening for client connections.

#### *ServerSocketChannel: Creating, accepting, and closing*

```
static ServerSocketChannel open()
ServerSocket socket()
SocketChannel accept()
void close()
boolean isOpen()
```

A ServerSocketChannel is created by calling the open() factory method. Each instance wraps an instance of ServerSocket, which you can access using the socket() method. As illustrated in the earlier examples, you *must* access the underlying ServerSocket instance to bind it to a desired port, set any socket options, etc. After creating and binding, you are ready to accept client connections by calling the accept() method, which returns the new, connected SocketChannel. When you are finished with a ServerSocketChannel, call the close() method. As we've already mentioned, blocking channels provide little advantage over regular sockets, except that they can (must) be used with Buffers. You will therefore almost always be setting your channels to be nonblocking.

#### *SocketChannel,ServerSocketChannel: Setting blocking behavior*

```
SelectableChannel configureBlocking(boolean block)
boolean isBlocking()
```

To set a SocketChannel or ServerSocketChannel to nonblocking, call configureBlocking(false). The configureBlocking() method returns a SelectableChannel, the superclass of both SocketChannel and ServerSocketChannel.

Consider setting up a connection for a SocketChannel. If you give the open() factory method of SocketChannel a remote address, the call blocks until the connection completes. To avoid this, use the parameterless version of open(), configure the channel to be nonblocking, and call connect(), specifying the remote endpoint address. If the connection can be made without blocking, connect() returns true; otherwise, you need some way to determine when the socket becomes connected.

#### *SocketChannel: Testing connectivity*

```
boolean finishConnect()
boolean isConnected()
boolean isConnectionPending()
```

With a nonblocking SocketChannel, once a connection has been initiated, the underlying socket may be neither connected nor disconnected; instead, a connection is “in progress.” Because of the way the underlying protocol mechanisms work, the socket may persist in this state for an indefinite time. The finishConnect() method provides a way to check the status of an in-progress connection attempt on a nonblocking socket, or to block until the connection is completed, for a blocking socket. For example, you might configure the channel to be nonblocking, initiate a connection via connect(), do some other work, configure the channel back to blocking, then call finishConnect() to wait until the connection completes. Or you can leave the channel nonblocking and call finishConnect() repeatedly, as in TCPEchoClientNonblocking.java.

The isConnected() method allows you to determine whether the socket is connected so you can avoid having a NotYetConnectedException thrown (say, by read() or write()). You can use isConnectionPending() to check whether a connection has been initiated on this channel. You want to know this because finishConnect() throws NoConnectionPendingException if invoked when one hasn't been.

### **Selectors in Detail**

The example TCPEchoServerSelector shows the basics of using Selector. Here we consider some of the details.

#### *Selector: Creating and closing*

```
static Selector open()
boolean isOpen()
void close()
```

You create a selector by calling the open() factory method. A selector is either “open” or “closed”; it is created open, and stays that way until you tell the system you are finished with it by invoking its close() method. You can tell whether a selector has been closed yet by calling isOpen().

#### *Registering Interest in Channels*

Each selector has an associated set of channels which it monitors for specific I/O “operations of interest” to that channel. The association between a Selector and a Channel is represented by an instance of SelectionKey. The SelectionKey maintains information about the kinds of operations that are of interest for a channel in a *bitmap*, which is just an int in which individual bits have assigned meanings.

The possible operations of interest are defined by constants of SelectionKey class; each such constant is a *bitmask* with exactly one bit set.

#### *SelectionKey: Interest sets*

```
static int OP_ACCEPT
```

```

static int OP_CONNECT
static int OP_READ
static int OP_WRITE
int interestOps()
SelectionKey interestOps(int ops)

```

We specify an operation set with a bit vector created by OR-ing together the appropriate constants out of OP\_ACCEPT, OP\_CONNECT, OP\_READ, and OP\_WRITE. For example, an operation set containing read and write is specified by the expression (OP\_READ|OP\_WRITE). The interestOps() method with no parameters returns a bitmap in which each bit set indicates an operation for which the channel will be monitored. The other method takes such a bitmap to indicate which operations should be monitored. Any change to the interest set associated with a key (channel) does not take effect until the associated selector's select() method is next invoked.

#### *SocketChannel,ServerSocketChannel: Registering Selectors*

```

SelectionKey register(Selector sel, int ops)
SelectionKey register(Selector sel, int ops, Object attachment)
int validOps()
boolean isRegistered()
SelectionKey keyFor(Selector sel)

```

A channel is registered with a selector by calling the channel's register() method. At registration time we specify the initial interest set by means of a bitmap stored in an int; register() returns a SelectionKey instance that represents the association between this channel and the given selector. The validOps() method returns a bitmap indicating the set of valid I/O operations for this channel. For a ServerSocketChannel, accept is the only valid operation, while for a SocketChannel, read, write, and connect are valid. For a DatagramChannel() only read and write are valid. A channel may only be registered once with a selector, so subsequent calls to register() simply update the operation interest set of the key. You can find out if a channel is registered with any selector by calling the isRegistered() method. The keyFor() method returns the same SelectionKey that was returned when register() was first called, or if the channel is not registered with the given selector.

The following code registers a channel for both reading and writing:

```

SelectionKey key = clientChannel.register(selector,
    SelectionKey.OP_READ | SelectionKey.OP_WRITE);

```

#### *SelectionKey: Retrieving and canceling*

```

Selector selector()
SelectableChannel channel()
void cancel()

```

The Selector and Channel instances with which a key is associated are returned by its selector() and channel() methods, respectively. The cancel() method invalidates the key (permanently) and places it in the selector's *canceled set*. The key will be removed from all key sets of the selector on the next call to select(), and the associated channel will no longer be monitored (unless it is re-registered).

#### *Selecting and Identifying Ready Channels*

With our channels registered with the selector and the associated keys specifying the set of I/O operations of interest, we just need to sit back and wait for I/O. We do this using the selector.

#### *Selector: Waiting for channels to be ready*

```

int select()
int select(long timeout)
int selectNow()
Selector wakeup()

```

The select() methods all return a count of how many registered channels are ready for I/O operations in their interest set to be performed. (For example, a channel with OP\_READ in the interest set has data ready to be read, or a channel with OP\_ACCEPT has a connection ready to accept.) The three methods differ only in their blocking behavior. The parameterless method blocks until at least one registered channel has at least one operation in its interest set ready, or another thread invokes this selector's wakeup() method (in which case it may return 0). The form that takes a timeout parameter blocks until at least one channel is ready, or until the indicated (positive) number of milliseconds has elapsed, or another thread calls wakeup(). The selectNow() is a nonblocking version: it always returns immediately; if no channels are ready, it returns 0. The wakeup() method causes any invocation of one of the select methods that is currently blocked (i.e., in another thread) to return immediately, or, if none is currently blocked, the next invocation of any of the three select methods will return immediately.

After selection, we need to know which channels have ready I/O of interest. Each selector maintains a selected-key set containing the keys from the key set whose associated channels have impending I/O of interest. We access the selected-key set by calling the selectedKeys() method of the selector, which returns a set of SelectionKeys. We can then iterate over this set of keys, handling pending I/O for each:

```

Iterator<SelectionKey> keyIter = selector.selectedKeys().iterator();
while (keyIter.hasNext()) {
    SelectionKey key = keyIter.next();
    // ...Handle I/O for key's channel...
    keyIter.remove();
}

```

```
}
```

#### Selector: Getting key sets

```
Set<SelectionKey> keys()
Set<SelectionKey> selectedKeys()
```

These methods return the selector's different key sets. The `keys()` method returns *all* currently registered keys. The returned key set is immutable: any attempt to directly modify it (e.g., by calling its `remove()` method) will result in an `UnsupportedOperationException`. The `selectedKeys()` method returns those keys that were “selected” as having ready I/O operations during the last call to `select()`. The set returned by `selectedKeys()` is mutable, and in fact *must* be emptied “manually” between calls to `select()`. In other words, the select methods only *add* keys to the selected key set; they do not create a new set.

The selected-keyset tells us which channels have available I/O. For each of these channels, we need to know the specific ready I/O operations. In addition to the interest set, each key also maintains a set of pending I/O operations called its *ready set*.

#### SelectionKey: Find ready I/O operations

```
int readyOps()
boolean isAcceptable()
boolean isConnectable()
boolean isReadable()
boolean isValid()
boolean isWritable()
```

We can determine which operations in the interest set are available for a given key by using either the `readyOps()` method or the other predicate methods. `readyOps()` returns the entire ready set as a bitmap. The other methods allow each operation to be tested individually.

For example, to see if the channel associated with a key has a read pending we can either use:

```
(key.readyOps() & SelectionKey.OP_READ) != 0
```

or

```
key.isReadable()
```

The keys in a selector's selected-key set and the operations in each key's ready set are determined by `select()`. Over time, this information can become stale. Some other thread may handle the ready I/O. Also, keys don't live forever. A key becomes invalid when its associated channel or selector is closed. A key may be explicitly invalidated by calling its `cancel()` method. You can test the validity of a key by calling its `isValid()` method. Invalid keys are added to the cancelled-key set of the selector and removed from its key set at the next invocation of any form of `select()`, or `close()`. (Of course, removing a key from the key set means that its associated channel will no longer be monitored.)

#### Channel Attachments

When a channel is ready for I/O, we often need additional information to process the request. For example, with our Echo protocol, when a client channel is ready to write, we need data. Of course, the data we need to write was collected earlier by reading it from the same channel, but where do we store it until it can be written? Another example is the framing procedure. If a message arrives a few bytes at a time, we may need to store the parts received so far until we have the complete message. In both cases, we need to associate state information with each channel. Well, we're in luck! `SelectionKeys` make storing per-channel state easy with attachments.

#### SelectionKey: Find ready I/O operations

```
Object attach(Object ob)
Object attachment()
```

Each key can have one attachment, which can be any object. An attachment can be associated when the channel's `register()` method is first called, or added directly to the key later, with the `attach()` method. A key's attachment can be accessed using the `SelectionKey`'s `attachment()` method.

#### Selectors in a Nutshell

To summarize, here are the steps in using a Selector:

- I. Create a selector instance.
- II. Register it with various channels, specifying I/O operations of interest for each channel.
- III. Repeatedly:
  - 1 Call one of the select methods.
  - 2 Get the list of selected keys.
  - 3 For each key in the selected-keys set,
    - a. Fetch the channel and (if applicable) attachment from the key
    - b. Determine which operations are ready and perform them. If an accept operation, set the accepted channel to nonblocking and register it with the selector
    - c. Modify the key's operation interest set if needed
    - d. Remove the key from the selected-keys set

If selectors tell you when I/O is ready, do you still need nonblocking I/O? Yes. A channel's key in the selected-keys set doesn't guarantee

nonblocking I/O because key set information can become stale after select(). In addition, a blocking write blocks until all bytes are written; however, an OP\_WRITE in the ready set only indicates that at least one byte can be written. In fact, you cannot register a channel with a selector unless it is in nonblocking mode: the register() method of SelectableChannel throws an IllegalBlockingModeException if invoked when the channel is in blocking mode.

### Datagram (UDP) Channels

Java NIO provides datagram (UDP) channels with the DatagramChannel class. As with the other forms of SelectableChannel we've seen, a DatagramChannel adds selection and nonblocking behavior and Buffer-based I/O to the capabilities of a DatagramSocket.

#### DatagramChannel: Creating, connecting, and closing

```
static DatagramChannel open()
boolean isOpen()
DatagramSocket socket() void close()
```

A DatagramChannel is created by calling the open() factory method, which creates an unbound DatagramChannel. The DatagramChannel is simply a wrapper around a basic DatagramSocket. You may directly access the particular DatagramSocket instance using the socket() method. This will allow you to call basic DatagramSocket methods to bind, set socket options, etc. When you are finished with a DatagramChannel, call the close() method.

Once you create a DatagramChannel, sending and receiving is fairly straightforward.

#### DatagramChannel: Sending and receiving

```
int send(ByteBuffer src, SocketAddress target)
SocketAddress receive(ByteBuffer dst)
```

The send() method constructs a datagram containing the data from the given ByteBuffer and transmits it to the SocketAddress specifying the destination. The receive() method prepares to accept a datagram into the specified buffer and return the address of the sender. If the buffer's remaining space is smaller than the datagram, any excess bytes are silently discarded.

The following code segment creates a DatagramChannel and sends the UTF-16 encoded string "Hello" to a UDP server running on the same host on port 5000.

```
DatagramChannel channel = DatagramChannel.open();
ByteBuffer buffer = ByteBuffer.wrap("Hello".getBytes("UTF-16"));
channel.send(buffer, new InetSocketAddress("localhost", 5000));
```

The following code segment creates a DatagramChannel, binds the underlying socket to port 5000, receives a datagram with a maximum of 20 bytes, and converts the bytes to a string using UTF-16 encoding.

```
DatagramChannel channel = DatagramChannel.open();
channel.socket().bind(new InetSocketAddress(5000));
ByteBuffer buffer = ByteBuffer.allocateDirect(20);
SocketAddress address = channel.receive(buffer);
buffer.flip();
String received = Charset.forName("UTF-16").newDecoder().decode(buffer).toString();
```

In the send() example above, we don't explicitly bind to a local port so a random port is chosen for us when send() is called. The corresponding receive() method returns a SocketAddress, which includes the port.

If we're always going to send to and receive from the same remote endpoint, we can optionally call the connect() method and specify the SocketAddress of a remote endpoint.

#### DatagramChannel: Connecting DatagramChannels

```
DatagramChannel connect(SocketAddress remote)
DatagramChannel disconnect()
boolean isConnected()
int read(ByteBuffer dst)
long read(ByteBuffer[] dsts)
long read(ByteBuffer[] dsts, int offset, int length)
int write(ByteBuffer src)
long write(ByteBuffer[] srcts)
long write(ByteBuffer[] srcts, int offset, int length)
```

These methods restrict us to only sending to and receiving from the specified address. Why do this? One reason is that after connect(), instead of receive() and send(), we can use read() and write(), which don't need to deal with remote addresses. The read() and write() methods receive and send a single datagram. The scattering read, which takes an array of ByteBuffers, only receives a single datagram, filling in the buffers in order. The gathering write transmits a single datagram created by concatenating the bytes from all of the array buffers. The largest datagram that can be sent today is 65,507 bytes; attempts to send more will be silently truncated.

Another advantage of connect() is that a connected datagram channel may only receive datagrams from the specified endpoint so we don't have to test for spurious reception. Note that connect() for a DatagramChannel does nothing more than restrict send and receive endpoints; no packets are exchanged as they are for connect() on a SocketChannel, and there is no need to wait or test for the connection to be completed, as there is with a SocketChannel.

So far, DatagramChannels look a lot like DatagramSockets. The major difference between datagram channels and sockets is the ability of a channel to perform nonblocking I/O operations and use selectors. Selector creation, channel registration, selection, etc., work almost identically to the SocketChannel. One difference is that you cannot register for connect I/O operations, but you wouldn't want to, since a DatagramChannel's connect() never blocks anyway.

#### DatagramChannel: Setting blocking behavior and using selectors

```
SelectableChannel configureBlocking(boolean block)
boolean isBlocking()
SelectionKey register(Selector sel, int ops)
SelectionKey register(Selector sel, int ops, Object attachment)
boolean isRegistered()
int validOps()
SelectionKey keyFor(Selector sel)
```

These methods have the same behavior as for SocketChannel and ServerSocketChannel.

Let's rewrite our DatagramSocketUDP echo server using DatagramChannel. The server listens on the specified port and simply echoes back any datagram it receives. The main difference is that this server doesn't block on send() and receive().

#### UDPEchoServerSelector.java

```
0 import java.io.IOException;
1 import java.net.InetSocketAddress;
2 import java.net.SocketAddress;
3 import java.nio.ByteBuffer;
4 import java.nio.channels DatagramChannel;
5 import java.nio.channels.SelectionKey;
6 import java.nio.channels.Selector;
7 import java.util.Iterator;
8
9 public class UDPEchoServerSelector {
10
11     private static final int TIMEOUT = 3000; // Wait timeout (milliseconds)
12
13     private static final int ECHOMAX = 255; // Maximum size of echo datagram
14
15     public static void main(String[] args) throws IOException {
16
17         if (args.length != 1) // Test for correct argument list
18             throw new IllegalArgumentException("Parameter(s): <Port>");
19
20         int servPort = Integer.parseInt(args[0]);
21
22         // Create a selector to multiplex client connections.
23         Selector selector = Selector.open();
24
25         DatagramChannel channel = DatagramChannel.open();
26         channel.configureBlocking(false);
27         channel.socket().bind(new InetSocketAddress(servPort));
28         channel.register(selector, SelectionKey.OP_READ, new ClientRecord());
29
30         while (true) { // Run forever, receiving and echoing datagrams
31             // Wait for task or until timeout expires
32             if (selector.select(TIMEOUT) == 0) {
33                 System.out.print(".");
34                 continue;
35             }
36
37             // Get iterator on set of keys with I/O to process
38             Iterator<SelectionKey> keyIter = selector.selectedKeys().iterator();
39             while (keyIter.hasNext()) {
40                 SelectionKey key = keyIter.next(); // Key is bit mask
41
42                 // Client socket channel has pending data?
43                 if (key.isReadable())
44                     handleRead(key);
45             }
46         }
47     }
48
49     private void handleRead(SelectionKey key) {
50
51         ByteBuffer buffer = ((DatagramChannel) key.channel()).receive(key);
52         if (buffer.remaining() > 0) {
53             byte[] data = new byte[buffer.remaining()];
54             buffer.get(data);
55             DatagramChannel channel = ((DatagramChannel) key.channel());
56             channel.send(ByteBuffer.wrap(data), key.attachment());
57         }
58     }
59
60     private static class ClientRecord {
61         public void handleRead(SelectionKey key) {
62             System.out.println("Received message from " + key.attachment());
63         }
64     }
65 }
```

```

46     // Client socket channel is available for writing and
47     // key is valid (i.e., channel not closed).
48     if (key.isValid() && key.isWritable())
49         handleWrite(key);
50
51     keyIter.remove();
52 }
53 }
54 }
55
56 public static void handleRead(SelectionKey key) throws IOException {
57     DatagramChannel channel = (DatagramChannel) key.channel();
58     ClientRecord clntRec = (ClientRecord) key.attachment();
59     clntRec.buffer.clear(); // Prepare buffer for receiving
60     clntRec.clientAddress = channel.receive(clntRec.buffer);
61     if (clntRec.clientAddress != null) { // Did we receive something?
62         // Register write with the selector
63         key.interestOps(SelectionKey.OP_WRITE);
64     }
65 }
66
67 public static void handleWrite(SelectionKey key) throws IOException {
68     DatagramChannel channel = (DatagramChannel) key.channel();
69     ClientRecord clntRec = (ClientRecord) key.attachment();
70     clntRec.buffer.flip(); // Prepare buffer for sending
71     int bytesSent = channel.send(clntRec.buffer, clntRec.clientAddress);
72     if (bytesSent != 0) { // Buffer completely written?
73         // No longer interested in writes
74         key.interestOps(SelectionKey.OP_READ);
75     }
76 }
77
78 static class ClientRecord {
79     public SocketAddress clientAddress;
80     public ByteBuffer buffer = ByteBuffer.allocate(ECHOMAX);
81 }
82 }
```

### 3.2.13.5 Exercises

- 1      Modify TCPEchoClientNonblocking.java to use a fixed-length write buffer.
- 2      Write an echo client that uses Buffer and DatagramChannel.

## 3.3 Web Services

### 3.3.1 Web Services Architectures

#### 3.3.1.1 Definitions of Web Services

According to the World Wide Web Consortium (W3C, 2008), a web service is "a software system designed to support interoperable machine to machine interaction over a network." What you will see most often on the Web is an API that may be accessed over the Internet and executed on a remote system.

There are numerous definitions given for Web services, ranging from the highly technical ones to simplistic. For example, the W3C organization, which establishes the standards for Web services, goes in more details: "A Web service is a software system identified by a URI whose public interfaces and bindings are defined and described using XML. Its definition can be discovered by other software systems. These systems may then interact with the Web service in a manner prescribed by its definition, using XML-based messages conveyed by Internet protocols." A simpler definition, and perhaps more useful, might be: "a Web service is a software application, accessible on the Web (or an enterprise's intranet) through a URL, that is accessed by clients using XML-based protocols, such as Simple Object Access Protocol (SOAP) sent over accepted Internet protocols, such as HTTP. Clients access a Web service application through its interfaces and bindings, which are defined using XML artifacts, such as a Web Services Definition Language (WSDL) file."

According to Gartner research in 2001, "Web services are loosely coupled software components delivered over Internet standard technologies." In short, Web services are self-describing and modular business applications that expose the business logic as services over the Internet through programmable interfaces and using Internet protocols for the purpose of providing ways to find, subscribe, and invoke those services. Based on XML standards, Web services can be developed as loosely coupled application components using any programming language, any protocol, or any platform. This facilitates delivering business applications as a service accessible to anyone, anytime, at any location, and using any platform.

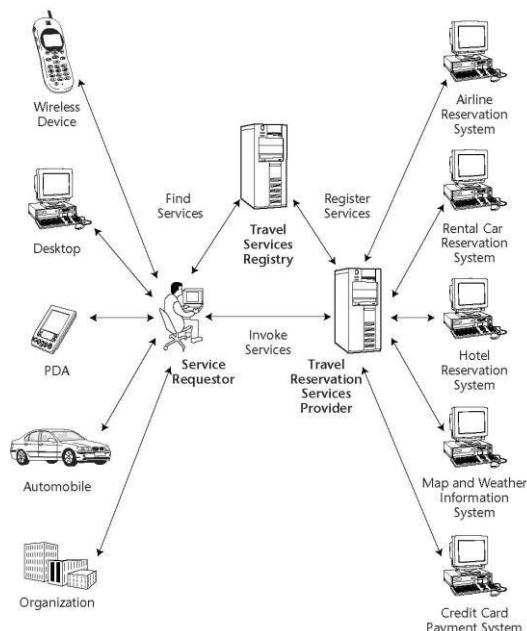


Fig. 3.14. An example scenario of Web services

Consider the example shown in Figure 3.14 where a travel reservation services provider exposes its business applications as Web services supporting a variety of customers and application clients. These business applications are provided by different travel organizations residing at different networks and geographical locations.

The realm of Web services—software components that are programmatically accessible over standard Internet protocols—is expanding rapidly due to the growing need for application-to-application communication and interoperability. Web services expose a standard interface that is platform and technology independent. By conforming to accepted industry-wide standards, Web services provide a means of communication among

software applications running on different platforms and written in different application development languages and that present dynamic context-driven information to the user.

### 3.3.3.2 Historical Evolution

Web Services have had a short but impressive history. In the late 1990s, Microsoft and a couple of other companies were thinking about an XML-based RPC that could work over HTTP. The term *SOAP* (Simple Object Access Protocol) was coined in 1998. The IETF published the first versions of SOAP 1.0 in December 1999. With broad support from both the commercial and Open Source community, a new version of SOAP emerged. In July 2001, the IETF published the first working draft of SOAP 1.2.

Microsoft, IBM, and Sun Microsystems are all pushing Web Services as the next great technology to allow developers to create remote objects easily. Earlier remote object technologies, such as COM+ and CORBA, were difficult to implement and had high maintenance costs. Additionally, in the case of CORBA, it was expensive to purchase the operational license. The promise of Web Services is to finally make remote objects a reality, but many of the details, such as security, seem to be hidden or spread across several different Web sites. Plus the term Web Services is generic and doesn't hint at all the underlying technologies.

Web services are a result of the natural evolution of the Web. Initially, the Web consisted of sites that were plain HTML pages. Later, Web applications dynamically generated these same HTML pages. For example, a map Web site initially provided only static links to maps of various cities and locales. Later, this same map Web site became a map Web application that provided driving directions, customized maps, and so forth. Despite their expanded capabilities, Web applications are still limited to the restricted GUI capabilities of their HTML pages—a Web application is usable only through the limited GUI bound to the HTML pages. Web services go beyond this limitation, since they separate the Web site or application (the service) from its HTML GUI. Instead, the service is represented in XML and available via the Web as XML. As a result, the same map Web site can extend its functionality to provide a Web service that other enterprises can use to provide directions to their own office locations, integrate with global position systems, and so forth.

Web services, or simply services, build on knowledge gained from more mature distributed computing environments (such as CORBA and Java Remote Method Invocation) to enable application-to-application communication and interoperability. Web services provide a standardized way for applications to expose their functionality over the Web or communicate with other applications over a network, regardless of the application's implementation, programming language, or computer platform.

Web services are one of the most significant advances in computing architecture over the past 30 years. Accordingly, it would be wise to become familiar with Web services architecture and concepts because it is highly likely that Web services will have a significant impact on the way enterprise conducts business in the future.

Today, people use the Internet as an everyday service provider for reading headline news, obtaining stock quotes, getting weather reports, shopping online, and paying bills, and also for obtaining a variety of information from different sources. These Web-enabled applications are built using different software applications to generate HTML, and their access is limited only through an Internet browser or by using an application-specific client. This is partially due to the limitations of HTML and the Web server-based technologies, which are primarily focused on presentation and their inability to interact with another application.

The emergence of Web services introduces a new paradigm for enabling the exchange of information across the Internet based on open Internet standards and technologies. Using industry standards, Web services encapsulate applications and publish them as services. These services deliver XML-based data on the wire and expose it for use on the Internet, which can be dynamically located, subscribed, and accessed using a wide range of computing platforms, handheld devices, appliances, and so on. Due to the flexibility of using open standards and protocols, it also facilitates Enterprise Application Integration (EAI), business-to-business (B2B) integration, and application-to-application (A2A) communication across the Internet and corporate intranet. In organizations with heterogeneous applications and distributed application architectures, the introduction of Web services standardizes the communication mechanism and enables interoperability of applications based on different programming languages residing on different platforms.

Traditionally, Web applications enable interaction between an end user and a Web site, while Web services are service-oriented and enable application-to-application communication over the Internet and easy accessibility to heterogeneous applications and devices.

### **3.3.1.3 Web Services Architecture as a Distributed Computing Architecture**

Web services are a distributed computing architecture—as are Common Object Request Broker (CORBA), Advanced Program-to-Program Communications (APPN), Electronic Data Interchange (EDI), and dozens of other preceding architectures.

The purpose of a distributed computing architecture is to enable programs in one environment to communicate and share data/content with programs in another environment. In the past, programmers have had to tell one application program where to go to find another cooperative program (known as "tightly coupling" applications). These programmers have had to maintain these programmatic links over the useful life of the applications that they have written. Creating these "hard-wired" links is complicated, cumbersome, and human resource intensive.

Now, imagine that a new distributed computing architecture has come along. And imagine that instead of requiring people (programmers) to establish and maintain program-to-program links, the applications themselves could automatically find cooperative programs to work with. And imagine that this new architecture allowed programmers to rapidly assemble complex applications merely by tying together application modules. And imagine that this new architecture allowed businesses to create or respond to competitive pressure more rapidly than ever before. And imagine that this new architecture allowed software to be delivered as a pay-as-you-go service. And so on.

Web services makes use of a program-to-program communications process called "loosely coupled." And by using this approach the amount of human involvement in building applications can be minimized. And the Web services protocols themselves are simpler and more straightforward to use than those of preceding architectures. Web services holds the promise of making programming vastly simpler.

But also consider this: loosely coupling applications has another important effect. Businesses can link strings of applications together to deliver new services to market—on the fly. And businesses can use Web services to market their existing nonstrategic software if they so desire. In short, Web services will change the way we build and use information systems—and will thus change the underlying business models upon which many enterprises currently operate.

Unstated but assumed in these brief definitions is that Web services are designed to be platform and language independent. This means that applications that use Web services are able to communicate with one another regardless of the underlying operating environment, system platform, or programming language being used. Applications can be written as discrete, self-contained "object modules"—application "blocks" that can service the needs of one application and, if appropriate, can also be reused to provide functions for other applications.

For instance, if a programmer were to write a calculator program, it could be written as a module and made available to a spreadsheet program, a customized transaction program, a mortgage amortization program, or any other program that could logically make use of a calculator. The point is that a calculator program has to be written only once (not constantly recreated) and can then be "bolted-onto," coupled, or reused with other applications in order to perform calculation services.

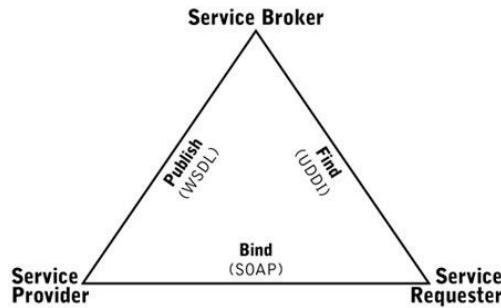
The use of "objects" is a fundamental concept in Web services, because it enables the assembly of large, compound applications faster than by today's monolithic methods. And, because programmers do not need to constantly recreate objects from scratch (they just plug in the appropriate object and away they go), application development will be less expensive.

### **3.3.1.4 Another Way of Defining What Web Services Do: Publish, Find, and Bind**

Here is another way to understand how Web services work. Web services directories and protocols essentially serve three functions: publish, find, and bind. To use Web services, applications are:

- Published in a common UDDI directory (such that cooperating applications can find each other);
- Found (using WSDL protocols that can locate Web services applications and determine if those applications can work with the source application); and
- Bound (a communications link between the two applications is established, so that a service—such as performing some sort of calculation or transaction—can be rendered using SOAP protocols).

These three service elements and their relationship to requester, broker, and provider services are illustrated in Figure 3.15.



**Figure 3.15. Web Services: Publish, Find, Bind.**

In this illustration a service requester is either you or your source application. The requester initiates the request for a Web service application. The Service Provider is the Web services application itself. The Service Broker can be a company that provides lists and information about Web services programs, or a Service Broker can refer to the programmatic process that helps locate an application and helps the two cooperating applications determine how to best communicate.

### 3.3.1.5 Types of Web Service Architectures

Web services are architected in different ways, and though they may vary in how they do their jobs, in the end they all get the job done. Because of the differences in web service architectures, applications must be designed with a specific type of web service in mind in order to utilize it effectively. The most common web service architectures are:

- Remote Procedure Call (RPC)
- Service-Oriented Architecture (SOA)
- Representational State Transfer (REST)

A **Remote Procedure Call (RPC)** architecture enables an application to start the process of an external procedure while being remote to the system that holds it. In simpler terms, a developer writes code that will call a procedure that could be executed either within the same application or in a remote environment. The developer does not care about the details of this remote action, only the interface it begins to execute and the results of that execution. The general concept of RPC dates back to the 1970s, when it was described in RFC 707. Not until the early 1980s, however, were the first implementations of RPC created. Microsoft used its version of RPC (MSRPC) as the basis for DCOM.

RPC fits the classic client/server paradigm for distributed computing. An RPC begins on the client by sending a request to a known remote server so that a specific procedure will be executed, with the client supplying parameters to do so. The code is then executed on the server, and a response is generated and returned back to the client. Here the original application continues to run as though the entire interaction is happening in a local environment.

A couple of popular variations on RPC exist in languages such as Java and Microsoft .NET. Java uses the Java Remote Method Invocation (Java RMI) to provide functionality similar to a standard RPC, whereas Microsoft has .NET Remoting to implement RPC for distributed systems in a Windows environment. XML-RPC provides a basic set of tools for creating cross-platform RPC calls, using HTTP as a foundation.

Web Services encapsulate *Remote Procedure Call (RPC)* with XML as the data packaging. The design of Web Services considers the pitfalls of the aforementioned remote object technologies and tries to avoid them. Although the use of Web Services does solve many of the problems, it also creates many new problems.

An alternative to RPC is to implement a Web Service with **Service-Oriented Architecture (SOA)** concepts, where applications are built with loosely coupled services. These services communicate using a formal definition (typically WSDL, discussed shortly) that is independent of the application's program language and the operating system in which it resides. The individual services are accessed without any knowledge of their underlying resource dependencies.

SOA has many definitions, and groups such as the Organization for the Advancement of Structured Information Standards (OASIS) and the Open Group have created formal definitions that can be applied to both technology and business. SOA adoption is thought to help the response time for changing market conditions, something that saves money in businesses. It also promotes reuse among components, a concept that is not new in programming circles. No matter what the belief, definition, or benefits, SOA has the following qualities:

- It is modular, interoperable, reusable, and component-based.
- It is standards-compliant.
- It has identifiable services, providing deliverables, with monitoring and tracking.

Combining SOA techniques with Web Services basically gives us the Web Services protocol stack, a collection of network protocols that are used to define and implement how Web Services interact with one another.

### 3.3.1.6 Elements of the Web Services Platform

Here are some of the key elements of a Web services platform:

- **Service contract** Unambiguous, well-defined service interface using WSDL. Ideally, it should be human-readable and machine-readable.
- **Service contract repository** A repository for storing, looking up, and versioning service contracts. This might include using taxonomies in UDDI or another registry to categorize services and then using these taxonomies to search for services; taxonomies give you much better searching capabilities than just searching based on the WSDL documents. Ideally, it should be highly available and replicated.
- **Service registration and lookup** A naming service for locating service instances and run-time resources in a high performance, scalable, and highly available manner. Whereas the service contract repository is used to look up service contracts, service registration and lookup is used for finding run-time instances of the services.
- **Service-level security** Security facilities using the WS-Security framework for defining and enforcing service-level security policies including authenticating service requesters, enforcing access control to service providers based on role and context authorization (e.g., role-based access control), single-sign-on, privacy, integrity, and non-repudiation. Of course, there will be other security facilities in the system for example, for controlling application-level user login and controlling database login but these facilities are not part of the Web services platform.
- **Service-level data management** XML Schema repository for storing and managing business-level data representations. Some organizations find it preferable to separate the service contract repository and the XML Schema repository because a single XML Schema definition can have many uses besides simply being included in a WSDL document, such as being used by data validation tools, data transformation tools, BPMS, reporting tools, rules engines, XML-relational mapping tools, and XML data servers. Data-mapping facilities for mapping data between different message structures including data filtering, data aggregation, and simple translation functions. Semantic-level data transform facilities define information taxonomies and perform semantic transformations across service domain boundaries.
- **Service-level communication** Support for multiple interaction patterns and communication styles using SOAP. Ideally, the communication infrastructure should support multiple interaction patterns and communication styles so that business requirements can be easily mapped onto the Web services platform.
- **Multiple protocol and transport support** Ideally, the messaging infrastructure should support multiple transports/protocols to support the wide range of clients, servers, and platforms.
- **Service-level qualities of service** Support for reliable messaging technologies and various qualities of service including message-ordering, guaranteed delivery, at-most-once delivery, and best-effort delivery. Transaction management capabilities for defining and supporting transaction execution and control including two-phase commit and/or compensating transactions. High-availability capabilities include clustering, failover, automatic-restart, load balancing, and hot-deployment of services.
- **Service-level management** Support for deploying, starting, stopping, and monitoring services. Of course, there will be other system management facilities in the system; for example, for managing hardware servers, but these facilities are not part of the Web services platform. Support for versioning services. Support for auditing service usage. Support for metering and billing for service usage. Service-level support for business activity monitoring (BAM) including service monitoring, service status, service responsiveness, and compliance or deviations from service-level agreements.
- **Support for multiple programming languages** Bindings for multiple programming languages. To fully support a wide range of applications and execution platforms, the Web services platform needs to support multiple programming languages, including generating service proxies and service skeletons for all supported programming languages.

- **Service programming interfaces** Typically, the Web services platform will provide service programming interfaces so that developers can access the facilities of the Web services platform from their favorite programming language(s) and so that developers can be isolated from the complexity of the underlying technical infrastructure.

### 3.3.1.7 Benefits of Web Services

Like any application, Web services-based applications can perform a range of functions. Some may handle only simple requests for information, while others may implement complex business processes and interactions. Whereas browser-based applications are concerned with the representation of data to end users, Web services let clients programmatically not only use the Web to obtain information but also to access these service components and their functionality. Furthermore, applications can incorporate Web service functionality for their own use.

Perhaps the most important reason for the increased use of Web services—the main force for their widespread adoption—is that Web services promote interoperability across different platforms, systems, and languages.

Use of Web services is also increasing because it reduces operational costs by enabling organizations to extend and reuse their existing system functionality.

Unlike traditional distributed environments, Web services emphasize interoperability. Web services are independent of a particular programming language, whereas traditional environments tend to be bound to one language or another. Similarly, since they can be easily bound to different transport mechanisms, Web services offer more flexibility in the choice of these mechanisms. Furthermore, unlike traditional environments, Web services are often not bound to particular client or server frameworks. Overall, Web services are better suited to a loosely coupled, coarse-grained set of relationships. Relying on XML gives Web services an additional advantage, since XML makes it possible to use documents across heterogeneous environments.

Web services, by building on existing Web standards, can be used without requiring changes to the Web infrastructure. However, while they may be more firewall friendly than traditional computing environments, Web services tend not to be as efficient in terms of space and time processing.

The following are the major technical reasons for choosing Web services over Web applications:

- Web services can be invoked through XML-based RPC mechanisms across firewalls.
- Web services provide a cross-platform, cross-language solution based on XML messaging.
- Web services facilitate ease of application integration using a light-weight infrastructure without affecting scalability.
- Web services enable interoperability among heterogeneous applications.

Web services are gaining in popularity because of the benefits they provide. Listed here are some of the key benefits:

- **Interoperability in a heterogeneous environment**— Probably the key benefit of the Web service model is that it permits different distributed services to run on a variety of software platforms and architectures, and allows them to be written in different programming languages. As enterprises develop over time, they add systems and solutions that often require different platforms and frequently don't communicate with each other. Later, perhaps due to a consolidation or the addition of another application, it becomes necessary to tie together this disparate functionality. The greatest strength of Web services is their ability to enable interoperability in a heterogeneous environment. As long as the various systems are enabled for Web services, they can use the services to easily interoperate with each other.
- **Business services through the Web**— An enterprise can use Web services to leverage the advantages of the World Wide Web for its operations. For example, an enterprise might make its product catalog and inventory available to its vendors through a Web service to achieve better supply chain management.
- **Integration with existing systems**— Most enterprises have an enormous amount of data stored in existing enterprise information systems, and the cost to replace these systems is such that discarding these legacy systems may not be an option. Web services let enterprise application developers reuse and even commoditize these existing information assets. Web services provide developers with standard ways to access middle-tier and back-end services, such as database management systems and transaction monitors, and integrate them

with other applications. In addition, because these services are provided consistently, developers do not need to learn new programming models or styles as integration needs expand.

- **Freedom of choice**— Web service standards have opened a large marketplace for tools, products, and technologies. This gives organizations a wide variety of choices, and they can select configurations that best meet their application requirements. Developers can enhance their productivity because, rather than having to develop their own solutions, they can choose from a ready market of off-the-shelf application components. Tools, furthermore, provide the ability to move quickly and easily from one configuration to another as required. Web services also ensure the standardization of tools, so that development tools can adopt new tools, whether from server vendors or third-party tool developers, as needs arise.
- **Support more client types**— Since a main objective of Web services is improving interoperability, exposing existing applications or services as Web services increases their reach to different client types. This occurs regardless of the platform on which the client is based: it doesn't matter if the client is based on the Java or Microsoft platforms or even if it is based on a wireless platform. In short, a Web service can help you extend your applications and services to a rich set of client types.
- **Programming productivity**— To be productive in the information economy requires the ability to develop and deploy applications in a timely fashion. Applications must go quickly from prototype to production and must continue to evolve even after they are placed into production. Productivity is enhanced when application development teams have a standard means to access the services required by multitier applications and standard ways to support a variety of clients. Web services, by creating a common programming standard, help to enhance programming productivity. Prior to the advent of Web services, developers programming in the distributed computing environment have relied on a diverse set of not-always-compatible technologies. Developers have attempted to tie together various diverse back-end systems, such as both custom and standard database management systems and transaction processors, with traditional Web technologies, but have had to deal with a multitude of programming models. Because Web services introduce a common standard across the Web, vendors, in the interest of staying competitive, are more likely to develop better tools and technologies. These tools and technologies will attract developers because they emphasize increased programming productivity. As a result, the entire industry benefits.

Application development has also been complicated by the requirement that a particular application support a specific type of client or several types simultaneously. Web services, because they promote interoperability, simplify this facet of the application development process.

### 3.3.1.8 Defining Objects and Web Services

The introduction of object-oriented programming promised the reusability of code across multiple systems and architectures. Many predicted that programmers eventually would not need to create their own objects because code repositories would possess all the necessary code in class files.

Some vendors took advantage of this and provided class libraries that took care of basic functionality. For example, many commercial libraries in C++ contain standard ways of dealing with date and time. This saves the developer time because the date and time functionality is already written in a standard way. When newer languages emerged, such as C# and Java, this basic functionality came as part of the language.

However, the concept of a central repository does not work in every situation. For example, because business logic varies from corporation to corporation, it would be impossible for a repository to contain that code. In this case, commercial libraries can only help by providing basic and generic classes that are commonly needed, but these libraries cannot always contain the necessary logic that a corporation needs because its situation is so unique. Certain business situations, such as in the banking industry, allow vendors to create prepackaged libraries because government regulations enforce certain logic.

At first, objects were only available to programs running on the same machine. For example, on the *Windows* directory of a PC there are several *Dynamic Link Libraries* (dll) files that contain information for *Windows* and other programs to operate. Each of these dlls contains information that several applications use to create objects. The dlls reside in the *Windows* directory so applications can find them without much work and different applications can then share the same dll.

Using objects in software is a great way of reusing code at the application level on a local machine. If you're in a corporation and you're maintaining several hundred or thousand workstations and the applications on each

system, having these libraries on every system becomes a maintenance headache. If an update is needed, every system within a corporation needs the new code installed. Remote objects, which are objects instantiated on a central server and can be accessed throughout a network including the Internet, become critical at this point.

A remote object is available to a program across a network or even the Internet. *Common Object Manifest* (COM), *Common Object Request Brokerage* (CORBA), *Remote Method Invocation* (RMI), and *Remote Procedure Call* (RPC) all invoke objects remotely. The promise of each of these methods allows a developer to change code in one place and then all systems and applications using these objects instantly have access to the new code. The disadvantage here is that if you make a coding error, all the systems now access that error. This causes all the applications using that code to fail. A remote object can be shared across multiple applications.

Reaching the promise of remote objects has been difficult. Implementing CORBA takes a great deal of effort and a company incurs great expense to purchase the *Object Request Broker* (ORB). An ORB is the server needed for applications to use remote objects. Only a small percentage of programmers implement CORBA on a daily basis, so it is also difficult for a manager to hire someone who understands the technology.

COM and COM+ offer a somewhat simpler solution to remote object computing. Most of the software needed, such as *Visual Studio* and Microsoft *Transaction Server*, are relatively inexpensive and easily obtainable, but COM and COM+ are typically only available to *Windows* applications. So these objects are not available to applications running under *UNIX* or another non-Microsoft operating system. In addition, COM takes a highly skilled programmer who understands everything about this technology. Again, it is difficult to find a programmer at this level.

The quest for remote object technologies that are simpler to use and implement brought RMI and RPC into the forefront briefly. They are simply request and response systems whose function is similar to how a Web page responds to a request from a browser. The problem here lies with the implementation that each vendor decided to pursue, and thus working with RPC software from one vendor wasn't always compatible with another's. Again not having cross-platform compatibility prevented remote object technology's general acceptance.

In the late 1990s, Microsoft realized the weaknesses in its own remote object technology, COM and COM+. As they began their .NET initiative they saw that they needed something new in order for remote object technology to succeed. One of the aspects of remote objects Microsoft needed for success was the ability for the technology to be cross platform, and because all of Microsoft's products run under their own operating system, they needed a creative solution. They created an XML standard called the *Simple Object Access Protocol* (SOAP) which is the combination of an *eXtensible Markup Language* (XML) document and a standard protocol that can work across the Internet. The protocols SOAP uses to transmit data include the *Simple Transport Mail Protocol* (SMTP), the *File Transfer Protocol* (FTP), and the *Hypertext Transfer Protocol* (HTTP). Developers often refer to these as the Web service's protocol stack.

Once Microsoft's team perfected the SOAP standard, they open-sourced this technology by giving it to the *World Wide Web Consortium* (W3C). The W3C made SOAP an official standard so other vendors could start making similar remote object technologies that were compatible with the Microsoft technology. Web Service products that exist on other platforms, such as *UNIX* or *AS400*, are instantly compatible with Microsoft technologies running under various versions of *Windows*. This happened because Microsoft chose standard technologies such as XML and the various protocols that exist on all platforms.

Currently, with the release of .NET, many vendors seek to create Web Service products that work with Microsoft's. Soon it won't matter what platform or language you are using. With Web Services, remote objecting may finally reach its promise because Web Services are simple and standard and vendors are making them easier to implement.

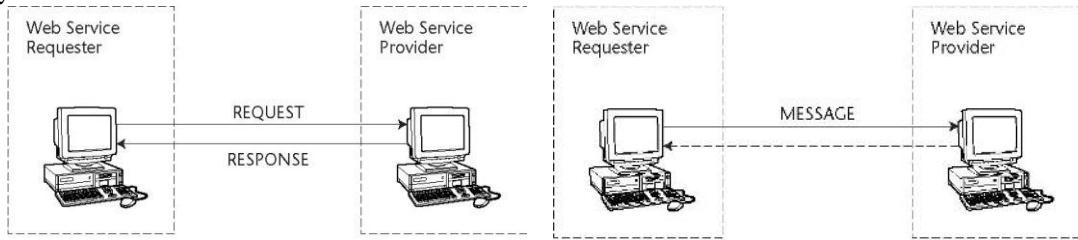
Before moving on, it is important to realize the difference between SOAP and a Web Service. SOAP is the actual protocol for moving data across the Internet. A Web Service is an object that uses SOAP to transmit data to an application or Web page. For example, a Web Service provides stock quotes whereas SOAP is that standard that allows the stock quote Web Service to be compatible across platforms.

### 3.3.1.9 Web Services Communication Models

In Web services architecture, depending upon the functional requirements, it is possible to implement the models

with RPC-based synchronous or messaging-based synchronous/asynchronous communication models. These communication models need to be understood before Web services are designed and implemented.

The **RPC-based communication model** defines a request/response-based synchronous communication. When the client sends a request, the client waits until a response is sent back from the server before continuing any operation. Typical to implementing CORBA or RMI communication, the RPC-based Web services are tightly coupled and are implemented with remote objects to the client application. Figure 3.16(a) represents an RPC-based communication model in Web services architecture. The clients have the capability to provide parameters in method calls to the Web service provider. Then, clients invoke the Web services by sending parameter values to the Web service provider that executes the required methods, and then sends back the return values. Additionally, using RPC-based communication, both the service provider and requestor can register and discover services, respectively.



**Fig. 3.16 (a) RPC-based communication model (b) Messaging-based communication model**

The **messaging-based communication model** defines a loosely coupled and document-driven communication. The service requestor invoking a messaging-based service provider does not wait for a response. Figure 3.16(b) represents a messaging-based communication model in Web services architecture. In Figure 3.16(b), the client service requestor invokes a messaging-based Web service; it typically sends an entire document rather than sending a set of parameters. The service provider receives the document, processes it, and then may or may not return a message. Depending upon the implementation, the client can either send or receive a document asynchronously to and from a messaging-based Web service, but it cannot do both functionalities at an instant. In addition, it also is possible to implement messaging with a synchronous communication model where the client makes the service request to the service provider, and then waits and receives the document from the service provider.

Adopting a communication model also depends upon the Web service provider infrastructure and its compliant protocol for RPC and Messaging. The current version of SOAP and ebXML Messaging support these communication models; it is quite important to ensure that the protocols are compliant and supported by the Web services providers. It also is important to satisfy other quality of services (QoS) and environmental requirements like security, reliability, and performance.

### 3.3.1.10 Overview of Web Service Standards

Standards differ from technologies. Standards are a collection of specifications, rules, and guidelines formulated and accepted by the leading market participants. While these rules and guidelines prescribe a common way to achieve the standard's stated goal, they do not prescribe implementation details. Individual participants devise their own implementations of an accepted standard according to the standard's guidelines and rules. These various implementations of a standard by different vendors give rise to a variety of technologies. However, despite the implementation detail differences, the technologies can work together if they have been developed according to the standard's specifications.

For Web services to be successful, the Web service standards must be widely accepted. To enable such wide acceptance, the standards used for Web services and the technologies that implement those standards should meet the following criteria:

- A Web service should be able to service requests from any client regardless of the platform on which the client is implemented.
- A client should be able to find and use any Web service regardless of the service's implementation details or the platform on which it runs.

Standards establish a base of commonality and enable Web services to achieve wide acceptance and interoperability. Standards cover areas such as:

- Common markup language for communication— To begin with, service providers, who make services available, and service requestors, who use services, must be able to communicate with each other. Communication mandates the use of a common terminology, or language, through which providers and requestors talk to one another. A common markup language facilitates communication between providers and requestors, as each party is able to read and understand the exchanged information based on the embedded markup tags. Although providers and requestors can communicate using interpreters or translators, using interpreters or translators is impractical because such intermediary agents are inefficient and not cost effective. Web services use eXtensible Markup Language (XML) for the common markup language.
- Common message format for exchanging information— Although establishing a common markup language is important, by itself it is not sufficient for two parties (specifically, the service providers and service requestors) to properly communicate. For effective communication, the parties must be able to exchange messages according to an agreed-upon format. By having such a format, parties who are unknown to each other can communicate effectively. Simple Object Access Protocol (SOAP) provides a common message format for Web services.
- Common service specification formats— In addition to common message formats and markup language, there must be a common format that all service providers can use to specify service details, such as the service type, how to access the service, and so forth. A standard mechanism for specifying service details enables providers to specify their services so that any requestor can understand and use them. For example, Web Services Description Language (WSDL) provides Web services with common specification formats.
- Common means for service lookup— In the same way that providers need a common way to specify service details, service requestors must have a common way to look up and obtain details of a service. This is accomplished by having common, well-known locations where providers can register their service specifications and where requestors know to go to find services. By having these common, well-known locations and a standard way to access them, services can be universally accessed by all providers and requestors. Universal Description, Discovery, and Integration (UDDI) specification defines a common means for looking up Web services.

The Web services standards (SOAP, WSDL, UDDI, WS-\*) are the best bet for defining a service platform, but it may be several years before the complete set of standards is approved, implemented, and interoperable across most vendor products. Because the Web services specifications are composable, however, it's possible to start with what's available now and add the rest later. In most cases, the platform will be composed of a wide assortment of products from a number of vendors almost by definition, it can't come from a single vendor.

### 3.3.1.11 Web Services Protocol Stack

Web services:

- Are designed to enable application modules (objects) to communicate with other application modules. Once connected, service applications provide transactional or computational services.
- Make use of a common data/information formatting scheme known as eXtensible Markup Language (XML) to share data.
- Applications make use of certain Web standards (UDDI, WSDL, and SOAP) for registry and program-to-program communications purposes.
- Use the Internet as the common network backbone.

The web services protocol stack has four basic levels:

- Service Transport
- Service Messaging
- Service Description
- Service Discovery

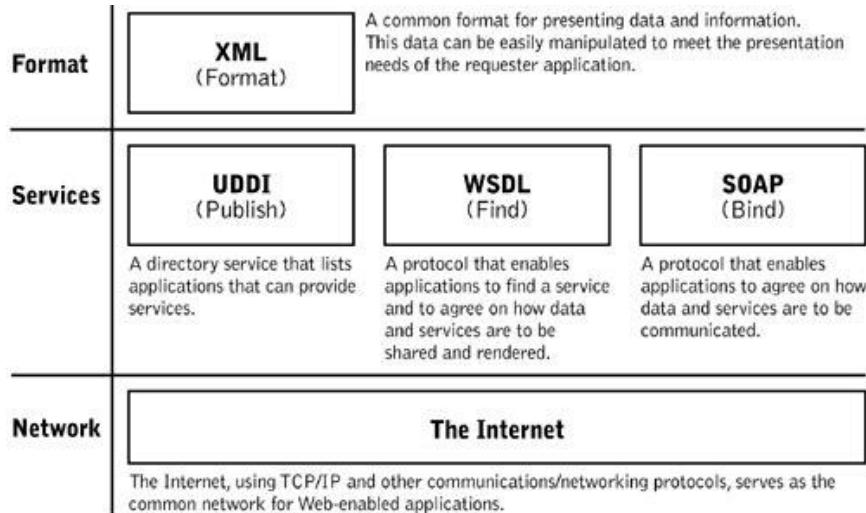
In addition to these basic levels, there are other important levels to consider, including Service Security and Service Management.

The Service Transport level is responsible for getting web services data between networked applications. Protocols involved at this level include HTTP or HTTPS, SMTP, and FTP. Service Messaging controls encoding messages in an XML format that is understood at both ends of a connection. Protocols for this level are XML-

RPC and SOAP. A description of the service must exist, and at the Service Description level, the WSDL format is usually used. At the Service Discovery level, a common repository for web services is used to publish their locations and descriptions. This makes it easier to find the services available on a network, and the UDDI protocol is used for this purpose.

These levels create the foundation on which Web Services that follow an SOA are built. SOA implementations rely on several standards to implement Web Services. These include XML, HTTP/HTTPS, SOAP, WSDL, and UDDI. A system does not have to have all of these standards to be considered an SOA, however.

Web services pass content between applications using a common format known as XML; Web services use a registry (UDDI), a template (WSDL), and a programmatic interface (SOAP) to enable applications to find and interact with each other; and they use a common network (the Internet) to transport information and data between cooperating applications (see Figure 3.17).



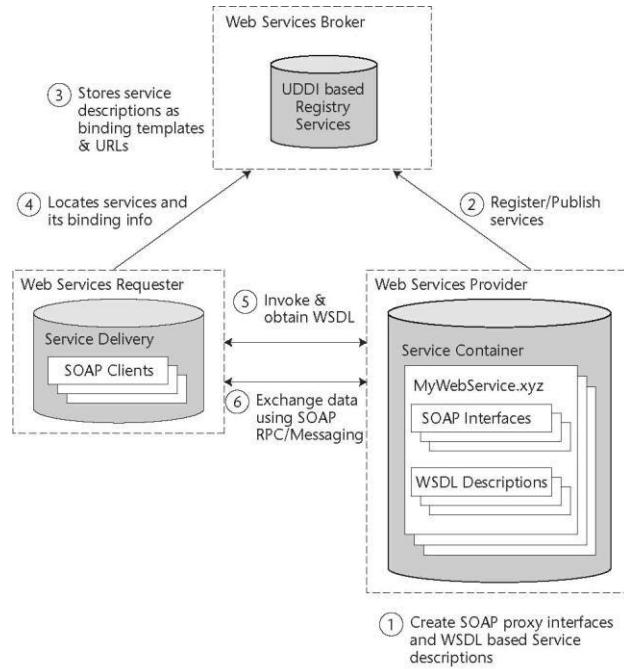
**Fig. 3.17. Critical Elements of a Basic Web Services Architecture.**

These are considered to be the basic building blocks of Web services architecture. But, as stated previously, the benefits of Web services architecture can be maximized if used in conjunction with business process management software to help streamline business workflow. And, by making application program development easier and linking application development with Web services protocols, more Web services applications can be developed. The next section describes how application development environments, Web (portal) development environments, mobile computing, and business process management software can all be used to create a highly integrated Web services development environment.

The process of **implementing Web services** is quite similar to implementing any distributed application using CORBA or RMI. However, in Web services, all the components are bound dynamically only at its runtime using standard protocols. Figure 3.18 illustrates the process highlights of implementing Web services. As illustrated in Figure 3.18, the basic steps of implementing Web services are as follows:

1. The service provider creates the Web service typically as SOAP-based service interfaces for exposed business applications. The provider then deploys them in a service container or using a SOAP runtime environment, and then makes them available for invocation over a network. The service provider also describes the Web service as a WSDL-based service description, which defines the clients and the service container with a consistent way of identifying the service location, operations, and its communication model.
2. The service provider then registers the WSDL-based service description with a service broker, which is typically a UDDI registry.
3. The UDDI registry then stores the service description as binding templates and URLs to WSDLs located in the service provider environment.
4. The service requestor then locates the required services by querying the UDDI registry. The service requestor obtains the binding information and the URLs to identify the service provider.
5. Using the binding information, the service requestor then invokes the service provider and then retrieves the WSDL Service description for those registered services. Then, the service requestor creates a client proxy application and establishes communication with the service provider using SOAP.

6. Finally, the service requestor communicates with the service provider and exchanges data or messages by invoking the available services in the service container.



**Fig. 3.18 Process steps involved in implementing Web services**

**Metadata management** includes the description information about a Web service necessary to construct a message body (including its data types and structures) and message headers so that a service requester can invoke a service. The provider of the service publishes the metadata so that a requester can discover it and use it to construct messages that can be successfully processed by the provider.

When invoking a service, it's important to understand not only the data types and structures to send but also the additional qualities of service provided (if any), such as security, reliability, or transactions. If one or more of these features are missing from the message, it may prevent successful message processing.

Metadata specifications include:

- XML Schema For message data typing and structuring and expressing policy information.
- WSDL For associating messages and message exchange patterns with service names and network addresses.
- WS-Addressing For including endpoint addressing and reference properties associated with endpoints. Many of the other extended specifications require WS-Addressing support for defining endpoints and reference properties in communication patterns.
- WS-Policy For associating quality of service requirements with a WSDL definition. WS-Policy is a framework that includes policy declarations for various aspects of security, transactions, and reliability.
- WS-MetadataExchange For querying and discovering metadata associated with a Web service, including the ability to fetch a WSDL file and associated WS-Policy definitions.

**Service binding** is different for an SOA based on Web services compared to an SOA based on J2EE or CORBA, for example. Instead of binding via reference pointers or names, Web services bind using discovery of services, which may be dynamic. If the service requester can understand the WSDL and associated policy files supplied by the provider, SOAP messages can be generated dynamically to execute the provider's service. The various metadata specifications are therefore critical to the correct operation of an SOA based on Web services.

Supposedly, the composability of extended Web services specifications allows their incremental use or "progressive discovery" of new concepts and features. IBM and Microsoft, as the de facto leaders of the Web services specifications development, are obviously keen to avoid making them too complex. A large part – if not the largest part – of the value of Web services derives from their relative simplicity. CORBA, for example, is often criticized for being too complex and too hard to use. Of course, CORBA is easier than what preceded it, but CORBA is relatively complex compared to Web services. The issue with complexity is finding people well

enough trained to use the technology productively, and because the highest IT cost is still labor, complexity typically means additional project expense. So for Web services to keep their promises, and to avoid having the whole effort fall apart, IBM and Microsoft want to preserve simplicity even as they start adding complex features to the basic specifications. One argument is that Web services are designed to inherently support composition of new features, meaning existing applications can be extended instead of being changed. However, this is a relatively untested assertion because products that fully implement the extended specifications are not yet available, and it isn't at all clear that the products will implement the extended features in the same way. Furthermore, there is no overall architecture for Web services that defines how the extended features really work with each other.

### 3.3.1.12 Web Service Implementations

There appears to be two major technological camps in the Web Services industry. The first camp is Microsoft. Microsoft got a head start because its people developed the *SOAP* standard and then gave it to the open-source community. So before other developers knew about the *SOAP* standard, Microsoft had already begun developing programming languages such as C# and *Visual Basic.NET* to create a proprietary implementation.

The second camp in the Web Services industry revolves around Java, but it isn't only Sun Microsystems, the creator of Java, deploying technologies. In this case, several vendors are each plying for a piece of the marketplace. Some of these vendors include BEA, Cape Clear Software, IBM, and many more. There are several Web Service libraries that are free and easily downloaded. This includes technology from Sun and the Apache group.

The important thing to remember is that it's ok for there to be several vendors supporting Web Services. Unlike previous technologies, such as RPC, the underlying technologies are based on XML standards. So even though the functionality may be different, the use of these standards almost ensures compatibility across platforms.

**Microsoft Implementation.** As mentioned earlier in this section, Microsoft created a head start for itself by creating the *SOAP* standard; thus, its Web Services software is probably the easiest to use and deploy. Microsoft provides a large number of tools to make the creation and use of Web Services very easy. This includes the automatic generation of WSDL, discovery tools such as disco (which searches servers that have .NET Web services), browser-based testing and discovery of methods, and easy creation of Web Services within Microsoft's proprietary languages. In fact, it is just a matter of adding a few lines of information, not necessarily code, which makes an existing method a Web Service. Although easy to use and deploy, all of the Microsoft Web Service technologies rely on their Web server *Internet Information Server (IIS)*. While this is probably the easiest Web server to maintain and configure, it has also had its share of security problems.

**Java Implementation.** As the Java world sees the reaction to Web Services under .NET, more and more Web Services products start showing up. BEA, Sun, and IBM deliver more and more Web Service products. In addition, the Apache group provides a great free *SOAP* library to access Web Services. There are a lot of advantages to using Web Services with Java. First of all, several different vendors implement Web Services with Java. This gives you greater selection of products to choose from during implementation, and perhaps allows you to easily integrate Web Services into your already existing architecture. Java Web Services work on top of both *Java Server Pages (JSP)* and servlets, and a developer's choice for serving these technologies is far greater than with Microsoft's implementation. *Tomcat* which is a free Java server that integrates easily with *Apache*, is free from the *Apache* group. IBM's *Websphere*, BEA's *WebLogic*, and Sun's *iPlanet* server are all commercially available Web servers that allow a developer more options when deploying Web Services. This is unlike Microsoft's implementation where you are locked into one platform and one Web server.

**Other Technologies.** Java and .NET are not the only technologies that need Web Services. There are third-party products available that allow Web Services to integrate with CORBA, COBOL, C++, and other legacy systems. By using Web Services with these systems, a corporation saves money because legacy systems don't need to be replaced. Instead, by adding a Web Services layer on top of these systems, the information these systems contain is available to applications that consume Web Services.

Services that are to be exposed as Web Services have to be implemented in a specific programming language. Just like in CORBA, the question boils down to how the server-side mapping is defined for Web Services. More specifically, given a WSDL specification, what does the server-side mapping look like for a given programming

language.

The somewhat surprising fact is that Web Services have not standardized how to map WSDL to a given programming language. The important implication is that Web Services do not support portability of applications. This means that if a programmer writes a Web Service application, that application is closely locked in with the product used.

To demonstrate the lack of portability, we provide the implementation of the account example for two different Web Services platforms. Here is the implementation based on Sun's JDK:

```
import java.rmi.Remote;
import java.rmi.RemoteException;
public interface AccountIF extends Remote {
    public void deposit (int amount) throws RemoteException;
    public void withdraw (int amount) throws RemoteException;
    public int balance () throws RemoteException;
}
```

In Sun's version, a Web Service interface is first described by a Java interface. This Java interface has to extend the Remote interface, which is defined as part of the RMI (remote method invocation) package. The methods to be exposed as a Web Service have to be declared as methods in this interface. Each method has to be able to throw the exception RemoteException, which is also declared as part of the RMI package.

The following code excerpt demonstrates the implementation of the account interface based on BEA's WebLogic Server product:

```
public class Account implements com.bea.jws.WebService
{
    static final long serialVersionUID = 1L;
    /**
     * @common:operation
     */
    public void deposit (int amount);
    ...
}
```

The starting point of a Web Services implementation with BEA's product is a Java class and not an interface, unlike with Sun's JDK. This class implements a BEA-specific interface. Methods to be exposed through the Web Service have to have the special comment @common:operation. Furthermore, the methods do not necessarily need to throw an exception.

### 3.3.2 XML

Web pages are a wonderful source of information, but it takes a human to understand what they mean. Wouldn't it be great if all the information on the Web was available in a form that could be easily used by other programs? Think of the amazing applications that you could build.

In the early days of the Web, application developers tried to programmatically mine information from Web pages by *screen scraping*, a technique where HTML is parsed and its meaning is inferred based on assumptions about page layout, table headings, and other clues. Of course, screen scraping is a lost cause because Web designers change page layout frequently to keep their sites interesting.

The best way to make your information available to other programs on the Web is to publish it in XML format. To implement this approach, you'll have to either define an XML vocabulary that describes your application data or use an industry standard vocabulary if a suitable one exists. Indeed the predominant activity following the publication of the XML specification was the definition of standard vocabularies such as Mathematical Markup Language (MathML), Chemical Markup Language (CML), and even Meat and Poultry Markup Language (mpXML).

Although XML was initially touted as a “better HTML,” it soon became apparent that the real sweet spot for XML was as a data interchange format. The term *Web service* was coined to describe a Web application that exchanged information in XML format. The combination of HTTP and XML was extremely potent. HTTP had become ubiquitous on the Internet. Firewalls allowed HTTP traffic on port 80 to pass through while other protocols and ports were shut out.

XML was textual and architecturally neutral so there was no confusion about low-level details such as the order of bytes in an integer. Although more verbose than binary formats, XML became universally supported. All platforms had XML parsers. At last there was a protocol, HTTP, and a format, XML, that applications on any platform could use to communicate. Web services became the *lingua franca* for application integration over the Internet.

#### 3.3.2.1 Overview of XML

XML (eXtensible Markup Language) allows the structured representation of arbitrary data. Based on a standard representation of arbitrary data, program libraries for parsing and generating XML files facilitate the handling of XML data. XML files are simple text files that can be edited with any editor. XML is called a markup language because the data is “marked up” through what are called *tags* in XML. Here is a simple example of an XML specification:

```
<Person>
  <FirstName>Mickey</FirstName>
  <LastName>Mouse</LastName>
  <Age>75</Age>
</Person>
```

Person, FirstName, LastName, and Age are tags. A *start tag* is surrounded by “<” and “>” while an *end tag* is surrounded by “</” and “>”. Note that the identifiers Person, FirstName, and so on are application specific and are not part of the XML standard. One way to look at XML is that XML itself provides *Content* is the syntax of a language, and the applications-specific identifiers make up the vocabulary. Between the start tag and the end tag is the *content* of the tag. In our example Mickey is the content of the tag FirstName. The combination of start and end tags and the content is also referred to as an *element*.

Tags can be the content of other tags; for example, tag Age belongs to the content of tag Person. Tags have to be strictly nested, which results in a hierarchical or tree-like representation of the data. Tags can have one or more attributes, as illustrated by the next example:

```
<struct name="Person">
  <member type="string" name="first_name"/>
  <member type="string" name="last_name"/>
  <member type="int" name="age"/>
</struct>
```

Name and type are called *attributes*. Name is an attribute of tag struct. The value of an attribute is written between the double quotes. For example, Person is the value of attribute name. Technically, the values of the attributes belong to the content of a tag, but there are no absolute rules whether data should be placed as the content of a tag or as a value of an attribute of that tag. Note that if there is no content for a tag, the tag can be surrounded by “<” and “>” instead of an explicit end tag.

The previous example already provides a hint on how XML can be used in the context of a middleware. The previous XML could be interpreted as a type definition where Person is a structure with members first\_name, last\_name, and age. Each of those members has an associated type, as is common for programming languages. Note that the tags struct and member are specifically chosen for the context of representing a programming language data structure.

If the previous XML might be an example of a type definition, the first example presented in this section could be interpreted as an instance that conforms to the type definition. In this sense XML can be used to describe both the types and instances of data to be handled by a middleware.

### 3.3.2.2 Building an XML Document

To read an XML document, an application uses a *parser* to get the data contained in the document. A parser usually consists of a large API that allows the programmer to choose which elements to look at in the document. Microsoft’s .NET architecture provides a developer with several classes for accessing the data in a document, and the Apache group develops a parser called *Xerces*™ that works cross platform.

With Web Services, an application passes an XML document across the Internet with different transport protocols. Therefore, either a client or sever side program must parse the XML to get to the data within the document. The following sections describe many parts of an XML document that a parser encounters.

**Processing instruction.** The first part of any XML document is the *Processing Instruction* (PI). This tells the parser that the data is in an XML document and the version of XML used (at this point it’s always 1). The start of the document now looks like the following.

```
<?xml version="1.0" ?>
```

The version is always set to 1 because there hasn’t been another version of XML. This statement tells the parser where to begin looking for XML.

**Root element.** To have a useful document, data needs to be present. To begin describing data, a root element must be present. This is the outermost element in the document. An element is simply a tag that looks much like an HTML tag, but in the case of XML the programmer chooses the name of the tag. For this example, BOOK is the root element.

```
<?xml version="1.0" ?>
<BOOK>
</BOOK>
```

The element is the word BOOK surrounded by <>. The element with the slash, in this case </BOOK> is the closing element. An XML document must have only one root element, and this element must be the outermost element. Later you’ll see that the root element begins the definition of a *SOAP* document or a WSDL file.

**Empty elements.** With the small amount of data present in this document, the closing element isn’t really necessary. Using an *empty element*, which is an element with no closing tag, makes the data more succinct by just using / at the end. If we take the previous example and make BOOK an empty element, we have the following:

```
<?xml version="1.0" ?>
<BOOK TITLE="Distributed Systems"/>
```

**Attributes.** Additional information added to an element is an *attribute* that, in this case, is part of the opening BOOK element and it contains the title of a book. Attributes always appear as part of the opening element and can be in any element in the document (not just in the root element as in the examples thus far).

```
<?xml version="1.0" ?>
<BOOK TITLE="Cross Distributed Systems">
</BOOK>
```

The XML standard contains a great deal of flexibility because both elements and attributes are allowed. This gives you and the developers of an XML language, such as *SOAP*, great flexibility in design.

As shown in examples later in the chapter, attributes often define namespaces or locations, such as the next *SOAP* node, for the XML document. Be sure to see the definition of namespaces later in this chapter.

**Attribute Centric Data.** So far this document doesn't really give a user much information about the book. By adding more attributes to the document, a better definition of data occurs and the following is the possible result.

```
<?xml version="1.0" ?>
<BOOK TITLE="Distributed Systems"
      PAGECOUNT="400"
      AUTHOR="Ion Ionescu"
      PUBLISHER="New House Press"/>
```

This document is attribute centric because the information all resides within attributes.

**Element centric data.** Now the information is more descriptive, but another possibility is to format the data which elements that are children of BOOK, such as the following.

```
<?xml version="1.0" ?>
<BOOK>
  <TITLE>Distributed Systems</TITLE>
  <PAGECOUNT>400</PAGECOUNT>
  <AUTHOR>Ion Ionescu</AUTHOR>
  <PUBLISHER>New House Press</PUBLISHER>
</BOOK>
```

Because the data in this example belongs completely in elements, the document is considered element centric.

**Elements and attributes in the same document.** This example is *element centric* because all the data resides in elements and the previous example was *attribute centric* because the data resides in attributes. XML, however, does not require that a document be attribute or element centric because the data can be mixed, as shown in the following example.

```
<?xml version="1.0" ?>
<BOOK TITLE="Distributed Systems">
  <PAGECOUNT>400</PAGECOUNT>
  <AUTHOR>Ion Ionescu</AUTHOR>
  <PUBLISHER>New House Press</PUBLISHER>
</BOOK>
```

**Nested elements.** The elements chosen for this document only allow for one book to be in the document. If several books need to be in the document, more nesting needs to occur. By nesting BOOK under a different root element named LIBRARY, several occurrences of BOOK can occur in the document, as shown in the following example.

```
<?xml version="1.0" ?>
<LIBRARY>
  <BOOK TITLE="Distributed Systems">
    <PAGECOUNT>400</PAGECOUNT>
    <AUTHOR>Ion Ionescu</AUTHOR>
    <PUBLISHER>New House Press</PUBLISHER>
```

```

</BOOK>
<BOOK
    TITLE="Parallel Computing">
    <PAGECOUNT>500</PAGECOUNT>
    <AUTHOR>Paul Popescu</AUTHOR>
    <PUBLISHER>New House Press</PUBLISHER>
</BOOK>
</LIBRARY>

```

LIBRARY is now the root element. BOOK is a child of LIBRARY but is still the parent of PAGECOUNT, AUTHOR, and PUBLISHER. This document now has the ability to describe multiple books, and perhaps other items, that may fit within a LIBRARY such as a magazine.

**Using namespaces.** When considering XML schemas, it is important to understand the concept of XML namespaces. To enable using the same name with different meanings in different contexts, XML schemas may define a namespace. A *namespace* is a set of unique names that are defined for a particular context and that conform to rules specific for the namespace. Since a namespace is specific to a particular context, each namespace is unrelated to any other namespace. Thus, the same name can be used in different namespaces without causing a duplicate name conflict. XML documents, which conform to an XML schema and have multiple elements and attributes, often rely on namespaces to avoid a collision in tag or attribute names or to be able to use the same tag or attribute name in different contexts.

Namespaces ensure that the element names used in your XML document are unique. The namespace definition occurs in the root element (the outermost element) and utilizes a URL as a unique identifier. Realize that there is no required content at the URL. It's just an identifier that assists in making the elements unique. Defining the namespace occurs in the root element, as the following example illustrates.

```
<BOOK XMLNS:WEBSERVICES="www.newhouse.com/XML"></BOOK>
```

Then all the child elements of BOOK begin with the namespace.

```

<?xml version="1.0" ?>
<BOOK XMLNS:WEBSERVICES="www.newhouse.com/XML">
    <WEBSERVICES:TITLE>Distributed Systems</
        WEBSERVICES:TITLE>
    <WEBSERVICES:PAGECOUNT>400</WEBSERVICES:PAGECOUNT>
    <WEBSERVICES:AUTHOR>Ion Ionescu</WEBSERVICES:AUTHOR>
    <WEBSERVICES:PUBLISHER>New House Press</ WEBSERVICES:PUBLISHER>
</BOOK>

```

Namespaces are an important concept to understand because many of the XML standards underlying Web Services utilize them usually as a way to represent various elements that are vendor dependent or to support primitive types from schemas.

**Well-formed and valid XML.** An XML document is often referred to as being *well-formed* and *valid*. This means that the document meets all the rules and contains all the information specified in either a *Document Type Definition* (DTD) or in an XML schema. Both act as a packing slip for XML documents, specifying which data needs to be present in the document. Validating a document against a schema or a DTD is a costly process and, thus, probably only occurs during the development of *SOAP* software. Once a developer ensures that his software produces the correct XML in the *SOAP* transactions, the validation is probably turned off. This is all dependent on how each vendor implements Web Services.

A **well-formed XML** document follows the rules set forth by the W3C. Put simply, there must be one or more elements, there can only be one root element that is not overlapped by any other element, and every start tag must have an end tag unless it's an empty element. Thus, one of the original examples was valid when it just had one empty element.

```

<?xml version="1.0" ?>
<BOOK TITLE="Distributed Systems"/>

```

By removing the / at the end of the element, the BOOK element no longer has a closing / or element. Therefore, the following document is not well formed.

```
<?xml version="1.0" ?>
<BOOK TITLE="Distributed Systems">
```

But by adding a closing BOOK element, the document becomes well formed again.

```
<?xml version="1.0" ?>
<BOOK TITLE="Distributed Systems"></BOOK>
```

Another error that prevents a document from being well formed happens when the root element gets overlapped by another tag. In the following example, BOOKDATA overlaps the root element BOOK and this causes a parsing error.

```
<?xml version="1.0" ?>
<BOOK TITLE="Distributed Systems ">
  <AUTHOR> Ion Ionescu </AUTHOR>
  <BOOKDATA>
    <PAGECOUNT>400</PAGECOUNT>
    <PUBLISHER> New House Press </PUBLISHER>
  </BOOK>
</BOOKDATA>
```

A quick way of checking the well-formedness of a document is to have Internet *Explorer* view the document.

**A Document Type Definition (DTD) or XML Schema Definition (XSD)** describes the structure of an XML document. It has information on the tags the corresponding XML document can have, the order of those tags, and so forth. An XML document can be validated against its DTD or its XSD. Validating an XML document ensures that the document follows the structure defined in its DTD or XSD and that it has no invalid XML tags. Thus, systems exchanging XML documents for some purpose can agree on a single DTD or XSD and validate all XML documents received for that purpose against the agreed-upon DTD/XSD before processing the document.

**Validity and Document Type Definitions (DTDs)** are a hold over from the older *Serialized General Markup Language* (SGML) standard that the publishing industry created to publish books. They are slowly falling out of favor with developers because they do not use XML. Up until the recent time, DTDs were the only way to have a valid XML document, but they didn't provide many of the things needed for common programming such as types or order. They did, however, allow a user to specify entity references that gave the ability to substitute values in and out of XML documents. Consider the following simple XML document.

```
<?xml version="1.0" ?>
<BOOK TITLE="Distributed Systems">
  <PAGECOUNT>400</PAGECOUNT>
  <AUTHOR>Ion Ionescu</AUTHOR>
  <PUBLISHER>Editura de Vest</PUBLISHER>
</BOOK>
```

DTDs rarely occur outside the publishing industry. However, you will find that many tools, such as Sun Microsystems Forte, allow you to easily create them.

A simple DTD for this file would need to recognize that BOOK is the root element and TITLE, PAGECOUNT, AUTHOR, and PUBLISHER are all children of BOOK. Because BOOK is a root element, it cannot be optional but all the other elements can be. We also need the DTD to recognize that TITLE is an attribute of BOOK. The following is the appropriate DTD for this XML document.

```
<?xml version="1.0" ?>
<!-- This is a comment --&gt;
<!-- The following code is the DTD --&gt;
<!-- The PI and the DTD are the prolog of the document --&gt;
&lt;!DOCTYPE BOOK [
  &lt;!ELEMENT BOOK (PAGECOUNT?,AUTHOR+,PUBLISHER+)&gt;</pre>
```

```

<!ATTLIST BOOK TITLE CDATA #REQUIRED>
<!ELEMENT PAGECOUNT (#PCDATA) >
<!ELEMENT AUTHOR (#PCDATA) >
<!ELEMENT PUBLISHER (#PCDATA) >
]>
<BOOK TITLE="Distributed Systems">
  <PAGECOUNT>400</PAGECOUNT>
  <AUTHOR>Ion Ionescu</AUTHOR>
  <PUBLISHER>New House Press</PUBLISHER>
</BOOK>

```

The DTD at the beginning of the document is clearly not XML. It's a completely different language and doesn't provide many of the constructs needed to be useful to a developer. For example, it is not possible to specify the minimum or maximum number of AUTHORS that need to be in the document. Specifying quantities is done with the symbols +, \*, and ?. The + means 1 or more of the element whereas the \* means 0 or more. The ? means the element is optional.

The vagueness and the difficult syntax of DTDs cause most developers to look at schemas as a way to validate XML documents.

**XSD.** Unfortunately, DTDs are an inadequate way to define XML document formats. For example, DTDs provide no real facility to express data types or complex structural relationships. XML schema definitions standardize the format definitions of XML documents.

Code Example 3.23 shows the code from an XML document representing an individual's contact information.

### ***Example 3.23. XML Document Example***

```

<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<ContactInformation>
  <Name>Ion Ionescu </Name>
  <Address>
    <Street>B-dul Vasile Parvan 4</Street>
    <City>Timisoara</City>
    <State>Timis</State>
    <Country>RO</Country>
  </Address>
  <HomePhone>0256-592-316</HomePhone>
  <EMail>ion_ionescu@yahoo.com</EMail>
</ContactInformation>

```

Code Example 3.24. is the DTD for the XML document in Code Example DES.E1.

### ***Example 3.24. Document Type Definition***

```

<!ELEMENT ContactInformation (Name, Address, HomePhone, EMail)>
<!ELEMENT Name (#PCDATA)>
<!ELEMENT Address (Street, City, State, Country)>
<!ELEMENT Street (#PCDATA)>
<!ELEMENT City (#PCDATA)>
<!ELEMENT State (#PCDATA)>
<!ELEMENT Country (#PCDATA)>
<!ELEMENT HomePhone (#PCDATA)>
<!ELEMENT EMail (#PCDATA)>

```

Code Example 3.26 shows the XSD schema for the sample XML document in Code Example 3.25.

### ***Example 3.25. XML Document***

```

<?xml version="1.0" encoding="ISO-8859-1" standalone="yes"?>
<ContactInformation>

```

```

xmlns="http://simple.example.com/CInfoXmlDoc"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
xsi:schemaLocation=
    "http://simple.example.com/CInfoXmlDoc
     file:/CInfoXmlDoc.xsd">
<Name> Ion Ionescu </Name>
<Address>
    <Street>B-dul Vasile Parvan 4</Street>
    <City>Timisoara</City>
    <State>Timis</State>
    <Country>RO</Country>
</Address>
<HomePhone>0256-592-316</HomePhone>
<EMail> ion_ionescu@yahoo.com </EMail>
</ContactInformation>

```

**Example 3.26. XSD Schema**

```

<?xml version="1.0" encoding="UTF-8"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://simple.example.com/CInfoXmlDoc"
  xmlns=" http://simple.example.com/CInfoXmlDoc"
  elementFormDefault="qualified">
  <xsd:element name="ContactInformation">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="Name" type="xsd:string" />
        <xsd:element name="Address">
          <xsd:complexType>
            <xsd:sequence>
              <xsd:element name="Street" type="xsd:string" />
              <xsd:element name="City" type="xsd:string" />
              <xsd:element name="State" type="xsd:string" />
              <xsd:element name="Country" type="xsd:string" />
            </xsd:sequence>
          </xsd:complexType>
        </xsd:element>
        <xsd:element name="HomePhone" type="xsd:string" />
        <xsd:element name="EMail" type="xsd:string" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

*Note on namespaces.* Technically speaking, an XML namespace defines a collection of names and is identified by a URI reference. (Notice in Code Example 3.26 the code `xmlns="http://simple.example.com/CInfoXmlDoc"`. Code such as this indicates that the XML schema defines a namespace for the various elements and attributes in the document.) Names in the namespace can be used as element types or attributes in an XML document. The combination of URI and element type or attribute name comprises a unique universal name that avoids collisions. For example, in Code Example 3.26, there is a namespace that defines the `ContactInformation` document's element types, such as `Name` and `Address`. These element types are unique within the contact information context. If the document included another namespace context, such as `BankInformation` that defined its own `Name` and `Address` element types, these two namespaces would be separate and distinct. That is, a `Name` and `Address` used in the context of `BankInformation` would not conflict with a `Name` and `address` used in the context of `ContactInformation`.

**Validity and schemas.** The following XML code example is a schema generated by *Visual Studio.NET*. In the code, there are a lot of namespaces defined and many of them deal specifically with things *Visual Studio* needs, but there is still important information present in this code that helps a developer more than any DTD could.

Skip past the namespace definitions and look at the first `xs:element`. The first `xs:element` defines the requirement for `PAGECOUNT` in the XML document, and these requirements are that it is a string according to the `type` attribute, `minOccurs` set to 0 indicates that `PAGECOUNT` is not required, and `maxOccurs` means that

PAGECOUNT can appear a maximum of three times. Additionally, the xs:sequence tag allows you to determine the order of the elements in the schema.

```
<?xml version="1.0" ?>
<xs:schema id="NewDataSet"
    targetNamespace="http://www.newhouse.com/~vs1C0.xsd"
    xmlns:mstns="http://www.newhouse.com/~vs1C0.xsd"
    xmlns="http://www.newhouse.com/~vs1C0.xsd"
    xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:msdata="urn:schemas-microsoft-com:xml-msdata"
    attributeFormDefault="qualified"
    elementFormDefault="qualified">
<xs:element name="BOOK">
    <xs:complexType>
        <xs:sequence>
            <xs:element name="PAGECOUNT" type="xs:string" minOccurs="0"
                maxOccurs="3" msdata:Ordinal="0" />
            <xs:element name="AUTHOR" type="xs:string" minOccurs="0"
                msdata:Ordinal="1" />
            <xs:element name="PUBLISHER" type="xs:string" minOccurs="0"
                msdata:Ordinal="2" />
        </xs:sequence>
        <xs:attribute name="TITLE" form="unqualified" type="xs:string"/>
    </xs:complexType>
</xs:element>
<xs:element name="NewDataSet" msdata:IsDataSet="true"
    msdata:EnforceConstraints="False">
    <xs:complexType>
        <xs:choice maxOccurs="unbounded">
            <xs:element ref="BOOK" />
        </xs:choice>
    </xs:complexType>
</xs:element>
</xs:schema>
```

Schema types are often used to identify primitive types in a SOAP message.

The XML in a schema is quite complex and confusing, but rarely does a programmer need to worry about coding a schema by hand. There are several tools available, such as *Visual Studio.NET*, that generate schemas automatically based on a given XML document. The most a programmer might do is go in and modify any defaults such as minOccurs and maxOccurs.

It is important to notice that the schema does represent the primitive type that the value of the XML element must appear as. In this case, all the values comprise of xs:string but other types may be represented as well such as xs:int or xs:bool. Schema is one of the few XML standards that represent types in this manner.

Most developers favor schemas in software development. They provide many definitions, such as type and order, needed to generate code or to validate technical XML documents.

**Self-describing XML.** Consider the following data.

Distributed Systems, Ion Ionescu, 400, 2008

Before the advent of XML, transmitted data often looked like the previous data. A programmer would rely on another individual to state which field represented which data. The elements used in XML documents should describe the data they represent. When we convert the simple text data shown previously into XML, the results may look like the following.

```
<?XML VERSION="1.0"?>
<BOOK>
    <TITLE> Distributed Systems </TITLE>
    <AUTHOR> Ion Ionescu </AUTHOR>
```

```

<PAGECOUNT>400</PAGECOUNT>
<YEARPUBLISHED>2008</YEARPUBLISHED>
</BOOK>

```

In this example, the elements, such as `<TITLE></TITLE>`, describe the data represented. The XML standard allows a programmer to choose any element name needed to describe the data, but if the names chosen aren't descriptive, the following could be the result.

```

<?XML VERSION="1.0"?>
<THING>
  <NAME> Distributed Systems </NAME>
  <PERSON> Ion Ionescu </PERSON>
  <NUMBER>400</NUMBER>
  <YEAR>2008</YEAR>
</THING>

```

You no longer know that the data represents a book. Instead, it represents some sort of thing but you are no longer sure what type of thing it represents. In the previous example, you knew that the data represented information for a particular book.

You will see that *SOAP* messages are not really self-describing. Remember that these messages are meant for applications to read, so the message formats are not as clean as some of the other XML languages such as the *eXtensible Stylesheet Language* (XSL).

### 3.3.2.3 XML—Advantages and Disadvantages

The eXtensible Markup Language was designed by the W3C (World Wide Web Consortium) in 1998. It was designed exactly for data exchange purposes and has demonstrated its strength over time. XML is a simple, flexible, text-based markup language. XML data is marked using tags enclosed in angled brackets. The tags contain the meaning of the data they mark. Such markup allows different systems to easily exchange data with each other. This differs from tag usage in HTML, which is oriented to displaying data. Unlike HTML, display is not inherent in XML. The eXtensible Markup Language (XML), a standard accepted throughout the industry, enables service providers and requestors to communicate with each other in a common language. XML is not dependent on a proprietary platform or technology, and messages in XML can be communicated over the Internet using standard Internet protocols such as HTTP. Because XML is a product of the World Wide Web Consortium (W3C) body, changes to it will be supported by all leading players. This ensures that as XML evolves, Web services can also evolve without backward compatibility concerns.

The advantages that XML provides are significant. In fact, it is:

- Structured
- Portable
- Extensible
- Text format

The tree-based structure of XML may lead to some apparent problems. A common debate is about the fact that XML is not the best way to represent an arbitrary object because of its limitations when it comes to sharing object references. Imagine you have a number of customers from just one city, an overhead of redundant data (a number of identical "city" blocks) is possible, which is barely acceptable.

Indeed this example was made just to show a common misuse, where the attribute `<city>` should be considered as an independent **entity**, rather than a **value**.

A better approach would be to handle the problem just like you would do with a **relational database** that is moving repeated data outside of the main object and embedding just a reference to them in the latter.

With a **Stateful approach**, the client could have retrieved the list of all "city" entities at a previous stage. So when it calls the getAllCustomers service, this could return just the city ids.

#### Example 3.27. Stateful Approach

```

<Customers>
  <customer>
    <id>4</id>

```

```

<name>Smith Ltd</name>
<location>
    <address>39, Kensington Rd.</address>
    <city>LND</city>
</location>
</customer>
<customer>
    <id>7</id>
    <name>Merkx & Co.</name>
    <location>
        <address>39, Venice Blvd.</address>
        <city>LAX</city>
    </location>
</customer>
...
</Customers>

```

On the other side, if we want to adopt a **Stateless approach**, in order to have a **self-contained** service, we could embed in the response of all the needed lists of data.

### Example 3.28. Stateless Approach

```

<Entireresponse>
<cities>
    <city>
        <id>LND</id>
        <name>London</name>
        <country>UK</country>
    </city>
    <city>
        <id>LAX</id>
        <name>Los Angeles</name>
        <country>USA</country>
    </city>
</cities>

<customers>
    <customer>
        <id>4</id>
        ...
        <location>
            ...
            <city>LND</city>
        </location>
    </customer>
    <customer>
        <id>7</id>
        ...
        <location>
            ...
            <city>LAX</city>
        </location>
    </customer>
    ...
</customers>
</Entireresponse>

```

### **3.3.3 WSDL**

One consequence of the requirement to be accessible by other applications is that a Web service must provide a well-defined interface and, in practice, this interface is specified using a Web Service Description Language (WSDL) document. WSDL 1.1 is the most prevalent way to describe the interface of Web services.

The Web Services Description Language (WSDL) defines a standard way for specifying the details of a Web service. It is a general-purpose XML schema that can be used to specify details of Web service interfaces, bindings, and other deployment details. By having such a standard way to specify details of a service, clients who have no prior knowledge of the service can still use that Web service. WSDL describe what public methods are available and where the service is located.

The W3C Note for WSDL, called "Web Services Description Language (WSDL) 1.1" was made available in 2001. In 2002, a Working Draft of WSDL 1.2 was released. Though technically not a W3C Recommendation, WSDL is pretty much the universally accepted protocol for describing Web services.

WSDL 2.0 is a W3C Recommendation issued in 2007. WSDL 1.1 and 2.0 are conceptually very similar. However, WSDL 2.0 benefited from a long and careful review process, and it incorporates many new features that bring it better into line with Web architectural principles. The careful review and test process for WSDL 2.0 will undoubtedly also eliminate many of the interoperability problems that plagued WSDL 1.1. Of course, there is always inertia to overcome when a new specification like WSDL 2.0 seeks to replace a widely deployed incumbent like WSDL 1.1. WSDL 1.1 and WSDL 2.0 will coexist for a long time. New Web services that support Web architectural principles such as REST will probably be the first adopters of WSDL 2.0.

#### **3.3.3.1 Service Contracts**

Every service (i.e., line of business service or reusable technical service) has a well-defined, formal interface called its service contract that (a) clearly defines what the service does and (b) clearly separates the service's externally accessible interface from the service's technical implementation.

Elements of Service Contract:

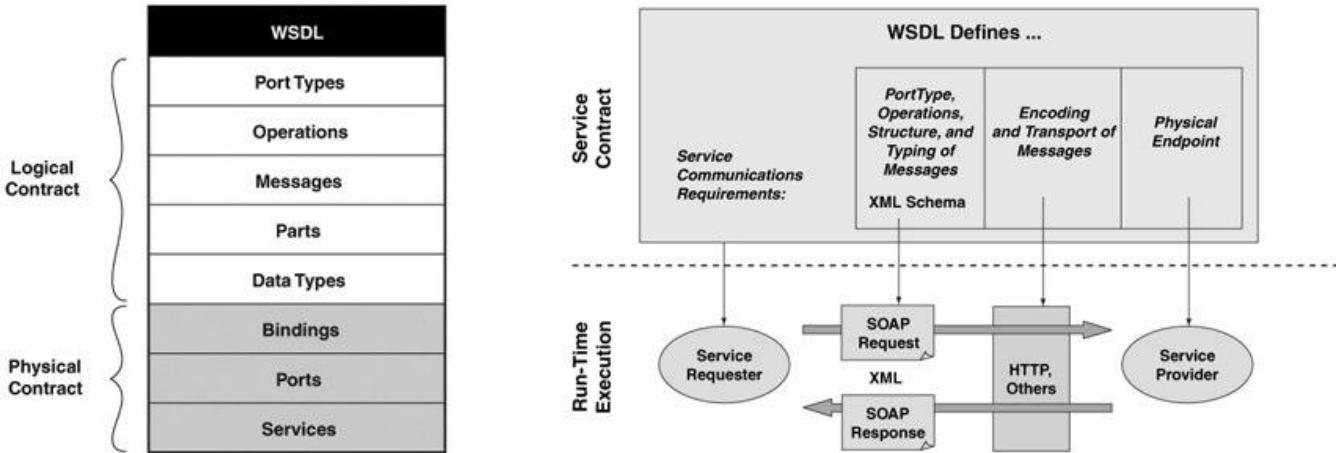
- Service names: Human-friendly name (plus aliases) as well as a unique machine-readable name.
- Version number: Supports the service lifecycle.
- Pre-conditions: Conditions that must be satisfied prior to using the service for example, an operation may not be accessible between midnight and 2 am.
- Service classification: Notes and keywords that identify the business domain(s) that the service supports "yellow page" entries for the service.

The service contract can be explicit and well-defined using WSDL, XML Schema, and the WS-Policy framework, or it can be implicitly defined based on the input messages the service accepts, the output messages it responds with, and the business activities that it implements. For Web services-based SOA implementations, WSDL is used to define key elements of the service contracts, while other elements that cannot be expressed in WSDL are defined using the WS-Policy framework or documented in a document or spreadsheet.

WSDL is the ideal choice as a service definition language because it is standards-based, extensible, built on XML Schema, and clearly separates the logical contract from the physical contract:

- The logical contract defines the public interface that is independent of transports, on-the-wire data formats, and programming languages.
- The physical contract defines bindings to transports and wire-level data formats, and multiple physical contracts can be defined for each logical contract.

The normal pattern for Web services interoperability is that one side creates an initial service, defining the messages, service names, and network address, and publishes the information within a WSDL file for external consumption. The requester of a Web service typically downloads or otherwise obtains a copy of the WSDL file and parses it to find out what kind of messages are required by the service provider, including the service name and associated data types and structures, and then it generates the message.



**Fig. 3.19 (a) WSDL structure (b) WSDL service contract architecture**

Figure 3.19(b) shows the major components of the WSDL service contract, including the XML Schema data types and structures for the messages, the specification of encoding and transport options, and the physical endpoint address for the service. These definitions can be used to dynamically generate the SOAP messages that are exchanged to execute the service, illustrated here using the request/response pattern.

The WSDL service contract defines the information necessary for interoperability (e.g., message format and transport details, such as HTTP, JMS, and WebSphere MQ) while at the same time defining the service in a manner that abstracts away (i.e., encapsulates) the execution environment (e.g., J2EE, .NET Framework, CORBA, CICS). Because J2EE, the .NET Framework, CORBA, and SAP NetWeaver (and virtually any other software system) all are capable of understanding Web services, achieving interoperability is a matter of defining and executing the appropriate Web services contract(s) upon which the disparate systems and applications can agree.

Web services contracts vary in complexity, including the following:

- Point-to-point agreements for solving specific interoperability problems, such as connecting J2EE to .NET.
- Complex messaging interactions such as publish and subscribe.
- Enterprise-wide patterns such as those designed for use in comprehensive integration architectures such as SOA.

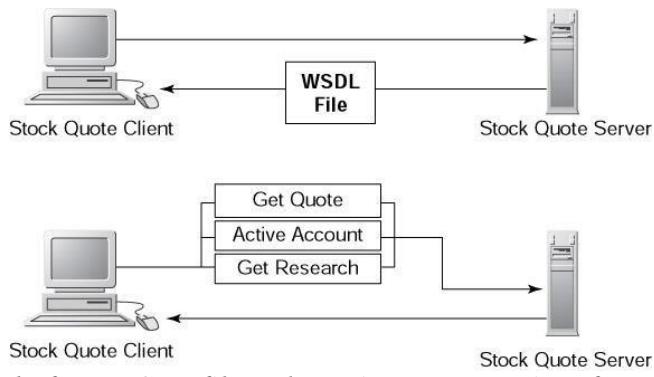
The basic Web services specifications provide interoperability solutions for sharing data across existing and new applications, while the extended Web services specifications provide integration solutions that include reliability, transactions, and security. WSDL service contracts can be easily extended to include these additional enterprise qualities of service typically present in existing applications.

### 3.3.3.2 WSDL in the Context of Remote Object Technologies

WSDL stands for *Web Services Description Language*, and it is a method of describing a Web Service using XML. For Web Services to work cross platform, they need to have a standard way to transmit and describe what happens within the Web Service. WSDL provides the means to describe what messages and variables exist within the Web Service whereas Universal Discovery, Description, and Integration (UDDI) is a standard way of publishing a Web Service to your desired audience. WSDL is part of the UDDI standard.

Having a standard way of describing a remote object is not unique to Web Services. CORBA has the *Interface Description Language* (IDL) that describes the CORBA object to the client, and Microsoft has the *Microsoft Interface Description Language* (MIDL) for COM+. IDL and, to a lesser extent, MIDL conform to the standard set by the Object Management Group (OMG). The OMG manages certain standards that the industry uses, just like the W3C, where particular aspects of a particular protocol are maintained by a committee and the results are published on the Web.

Figure 3.20 shows how a Web Service client uses a WSDL file. Even though this example is specific to Web Services, this interaction is similar for all remote object technologies.

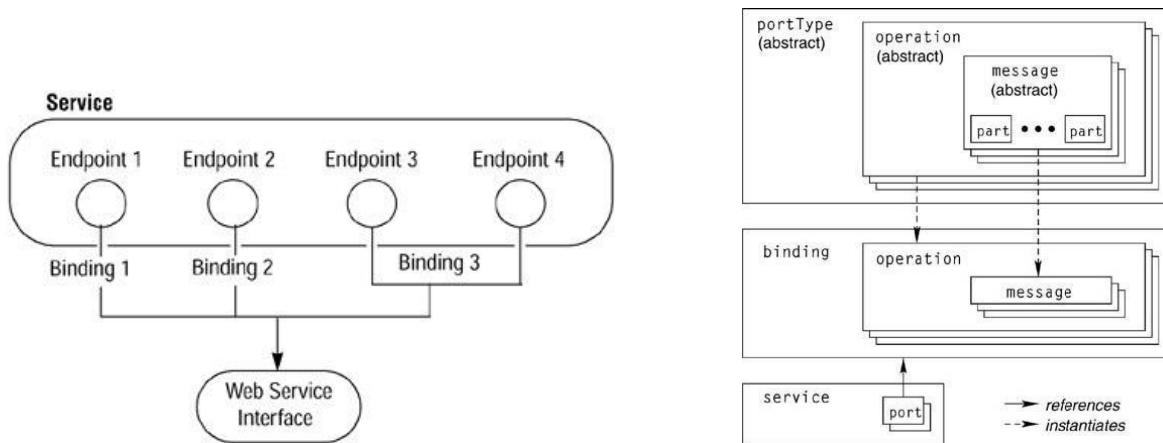


**Figure 3.20: How a client asks for a WSDL file and uses it to create an interface to the methods from the Web Service.**

Before any Web Service call is made, the user creating the client uses a tool to grab the WSDL file off the server; this is shown in the first transaction of Figure 3.20. Many Web Service technologies have a standard way of revealing their WSDL file so it isn't difficult for a user to find.

Once the user has the WSDL file, some technologies, such as Microsoft's .NET, allow the user to then create a proxy to the Web Service. This proxy then acts as an interface to the Web Service, allowing the client-side code to easily access the Web Service. In the case of Figure 3.20, the WSDL file reveals to the client that the Stock Quote server has three methods available: Get Quote, Active Account, and Get Research. The client now has an interface to each of these methods and is able to use them.

WSDL specifies a grammar that describes Web services as a collection of communication endpoints, called **ports**. The data being exchanged are specified as part of **messages**. Every type of action allowed at an endpoint is considered an **operation**. Collections of operations possible on an endpoint are grouped together into port types. The messages, operations, and port types are all abstract definitions, which means the definitions do not carry deployment-specific details to enable their reuse. The protocol and data format specifications for a particular port type are specified as a **binding**. A port is defined by associating a network address with a reusable binding, and a collection of ports define a service. In addition, WSDL specifies a common binding mechanism to bring together all protocol and data formats with an abstract message, operation, or endpoint. See Figure 3.21(a).



**Figure DES.F3. (a) WSDL Service Description (b) WSDL components**

WSDL resembles in its purpose CORBA's IDL. WSDL introduces a specific “vocabulary” for XML tags and attributes that allows the description of interfaces. Figure 3.21(b) provides a top-level overview of a WSDL specification.

While using XML to describe service interfaces has the benefit of not having to invent a new language, the downside is that XML specifications tend to get quite verbose. Web Services are promoting the idea that WSDL is generated from a programming language such as Java. These automatically generated WSDL specifications often have to be manually edited, so in most cases WSDL is not completely transparent to the applications programmer.

### 3.3.3.3 Structure of the WSDL Document

The WSDL document describes each piece of the Web Service from each of the elements found in the XML (besides the standard *SOAP* elements) to the transport and the name.

Six major elements are included in a WSDL document:

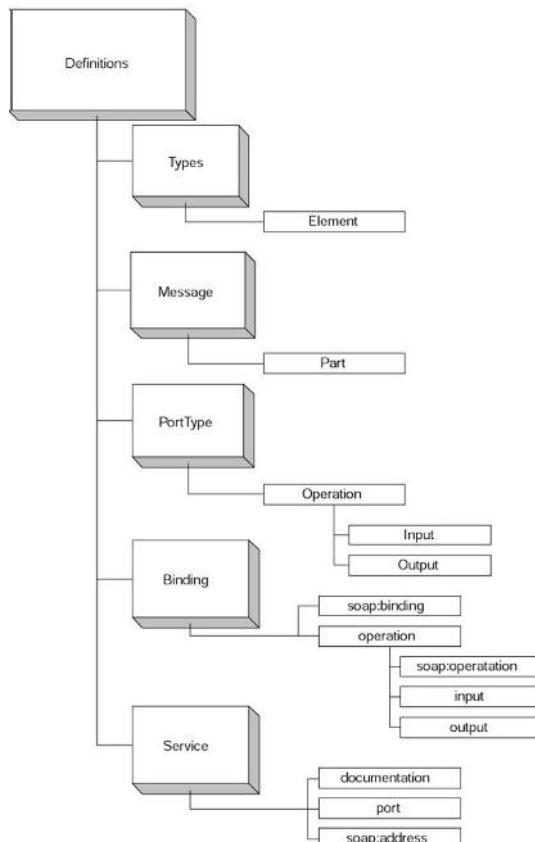
- The data types that the web service uses
- The messages that the web service uses
- The operations that the web service performs
- The communication protocols that the web service uses
- The individual binding addresses
- The aggregate of a set of related ports

The main six parts or elements of the WSDL document include: definitions, types, message, portType, binding, and service. Table 3.6 summarizes what each section describes.

**Table 3.6: The Parent Elements That Make up an XML Document**

Element	Purpose
definitions	The root element of the WSDL document. Defines many of the namespaces used for a particular description.
types	Defines the elements and primitive types found in the XML of the SOAP request and response.
message	Names the request and response messages.
portType	Ties a particular request and response message to a particular service.
binding	Indicates the type of transport (i.e., HTTP) used by the service. Also describes the contents of the message such as literal, meaning to take the XML at face value, or image/gif.
service	Actually names the services and provides an element to document what the service actually accomplishes.

To help illustrate the structure of the WSDL document, Figure 3.22 uses a diagram.



**Figure 3.22: Illustrates the structure of the WSDL document along with any child elements of each major section.**

You'll notice in Figure 3.23 that several of the elements actually have child elements as well. For example, the types element has an element that describes the elements found in the XML; it may contain several elements for a Web Service that contains many methods.

In summary, a port describes the *what* of a Web Service, the binding describes the *how*, and the service describes the *where*.

A WSDL document may contain other elements, and it can group together definitions of several web services into one WSDL document. Take a look at the W3C Note for more information on these elements. The structure of the document, according to the W3C Note, looks like this:

```
<wsdl:definitions name="nmtoken"? targetNamespace="uri"?>
    <import namespace="uri" location="uri"/>*
    <wsdl:documentation .... /> ?
    <wsdl:types> ?
        <wsdl:documentation .... />?
        <xsd:schema .... />*
        <!-- extensibility element --> *
    </wsdl:types>
    <wsdl:message name="nmtoken"> *
        <wsdl:documentation .... />?
        <part name="nmtoken" element="qname"? type="qname"?/>> *
    </wsdl:message>
    <wsdl:portType name="nmtoken">*
        <wsdl:documentation .... />?
        <wsdl:operation name="nmtoken">*
            <wsdl:documentation .... />?
            <wsdl:input name="nmtoken"? message="qname">?
                <wsdl:documentation .... />?
            </wsdl:input>
            <wsdl:output name="nmtoken"? message="qname">?
                <wsdl:documentation .... />?
            </wsdl:output>
            <wsdl:fault name="nmtoken" message="qname"> *
                <wsdl:documentation .... />?
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:portType>
    <wsdl:binding name="nmtoken" type="qname">*
        <wsdl:documentation .... />?
        <!-- extensibility element --> *
        <wsdl:operation name="nmtoken">*
            <wsdl:documentation .... />?
            <!-- extensibility element --> *
            <wsdl:input> ?
                <wsdl:documentation .... />?
                <!-- extensibility element -->
            </wsdl:input>
            <wsdl:output> ?
                <wsdl:documentation .... />?
                <!-- extensibility element --> *
            </wsdl:output>
            <wsdl:fault name="nmtoken"> *
                <wsdl:documentation .... />?
                <!-- extensibility element --> *
            </wsdl:fault>
        </wsdl:operation>
    </wsdl:binding>
    <wsdl:service name="nmtoken"> *
        <wsdl:documentation .... />?
        <wsdl:port name="nmtoken" binding="qname"> *
            <wsdl:documentation .... />?
            <!-- extensibility element -->
        </wsdl:port>
        <!-- extensibility element -->
```

```

</wsdl:service>
<!-- extensibility element --> *
</wsdl:definitions>

```

This may not mean a whole lot to anyone that does not enjoy reading through the entire specification for a piece of technology. Therefore, here is a brief explanation of the different parts.

**Definitions** is the root element of the WSDL document. The beginning of the document may look like the following.

```
<definitions name="GetStockQuote"></definitions>
```

This only defines the name of the Web Service. To be useful, however, the definitions tag will need to define several different namespaces to support the different primitive types and the types created by the GetStockQuote Web Service. Once the namespaces are added, the definitions element now looks like the following.

```

<definitions name="GetStockQuote"
    targetNamespace="http://advocatemedia.com/GetStockQuote.wsdl"
    xmlns:myns = "http://advocatemedia.com/GetStockQuote.wsdl"
    xmlns:myXsd = "http://advocatemedia.com/GetStockQuote.xsd"
    xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap"
    xmlns="http://schemas.xmlsoap.org/wsdl/">
</definitions>

```

Table 3.7 describes what each namespace definition does for the WSDL document.

**Table 3.7: Detailed Information about What Each Namespace Definition Does for the WSDL Document**

Namespace Definition	Placeholder	Description
targetNamespace	<a href="http://advocatemedia.com/GetStockQuote.wsdl">http://advocatemedia.com/GetStockQuote.wsdl</a>	Defines the namespace for this document.
Myns	<a href="http://advocatemedia.com/GetStockQuote.wsdl">http://advocatemedia.com/GetStockQuote.wsdl</a>	A more precise definition for this document.
MyXSD	<a href="http://advocatemedia.com/GetStockQuote.xsd">http://advocatemedia.com/GetStockQuote.xsd</a>	Namespace for the schema types defined here.
xmlns:soap	<a href="http://schemas.xmlsoap.org/wsdl/soap">http://schemas.xmlsoap.org/wsdl/soap</a>	Namespace for the <i>SOAP</i> elements used in the document.
xmlns	<a href="http://schemas.xmlsoap.org/wsdl/">http://schemas.xmlsoap.org/wsdl/</a>	Sets the default namespace for <i>SOAP</i> elements.

Remember that a namespace is just a unique placeholder for a set of elements in an XML document. The URL is just a unique value and is not necessarily a URL to which you may point your browser.

**Types.** The types element defines the different elements used within the Web Service. The elements defined here are for the variables in our Stock Quote Web Service. Just like with *SOAP*, WSDL uses the types from the schema standard so that standard types are used.

The types element defines a schema in the middle of our WSDL document. By doing this, a WSDL document is able to use the types defined in the schema standard rather than having to create its own.

Take a look at the following example and notice that several elements are defined.

```

<types>
    <schema
        targetNamespace="http://advocatemedia.com/GetStockQuote.xsd"
        xmlns="http://www.w3.org/2000/10/XMLSchema">
        <element name="StockQuoteRequest">
            <complexType>
                <all>
                    <element name="symbol" type="string"/>

```

```

        </all>
    </complexType>
</element>
<element name="StockQuoteResponse">
    <complexType>
        <all>
            <element name="price" type="float"/>
        </all>
    </complexType>
</element>
</schema>
</types>

```

StockQuoteRequest and StockQuoteResponse are the parent elements of the request and response documents. The elements that actually contain values (i.e., symbol and price), are the children. The types for these child elements also are defined here. symbol is defined as a string and price is defined as a float.

**Message.** Now the WSDL document needs to describe and name both the request and response message. This comes after the type definition and gives a path back to the types in the message. The message elements look like the following.

```

<message name="GetStockQuoteRequest">
    <part name="body" element="myXSD:StockQuoteRequest"/>
</message>
<message name="GetStockQuoteResponse">
    <part name="body" element="myXSD:StockQuoteResponse"/>
</message>

```

Notice that the element definitions have the names of the parent elements in the *SOAP* document. The namespace myXSD is used as a prefix so that the application using the WSDL document knows to find the definitions for these elements in the types portion of the document.

This gives the request and response messages a name so the applications or Web pages using this service know the name of the message to send and expect back when using a particular service. Note that this is protocol independent because there is no mention of HTTP or SMTP.

The value of the name can be anything, one should select something that is meaningful.

**PortType.** To use the two messages defined in the previous section with the message element, you must define them as the request and response for a particular service.

This is done with the portType command, as shown in the following example.

```

<portType name="GetStockQuotePort">
    <operation name="GetStockQuote">
        <input message="myns:GetStockQuoteRequest"/>
        <output message="myns:GetStockQuoteResponse"/>
    </operation>
</portType>

```

GetStockQuote is now considered the name of the operation. The operation is the request for the price of a particular stock and the response is in the form of a price for that stock. The input and output messages just combine the two definitions used earlier so that the client knows that a particular request and response message belongs to a particular method in a Web Service.

If this were a more complex Web Service, there would be several portType, message, and other elements. The example we're using here is very simple so the WSDL isn't too complex.

A portType is an abstract definition of an interface. It is abstract in the sense that it describes the operational interface of a service without going into the details of the data layout of the various parameters. A portType essentially consists of one or more operations, each consisting of several messages. By explicitly defining

messages for each operation, it is possible to do interactions other than RPC-style operations. For example, a notification would only consist of one message, while an RPC-style operation would consist of two messages (request and response). The signature of a message is defined through a sequence of part elements, each describing one formal input/output parameter. The following XML excerpt shows the WSDL specification for our account example. Note that the XML has been simplified for readability purposes:

```

<definitions name="MyAccountService">
  <types/>
  <message name="AccountIF_balance"/>
  <message name="AccountIF_balanceResponse">
    <part name="result" type="int"/>
  </message>
  <message name="AccountIF_deposit">
    <part name="amount" type="int"/>
  </message>
  <message name="AccountIF_depositResponse"/>
  <message name="AccountIF_withdraw">
    <part name="amount" type="int"/>
  </message>
  <message name="AccountIF_withdrawResponse"/>
  <portType name="AccountIF">
    <operation name="deposit" parameterOrder="amount">
      <input message="AccountIF_deposit"/>
      <output message="AccountIF_depositResponse"/>
    </operation>
    <!--similar definitions for withdraw and balance -->
  </portType>

```

One interesting fact to note is that unlike the XML tag portType might suggest, it does not introduce a new type. For example, it is not possible to use AccountIF as defined above as a type of a formal parameter of an operation. The implication is that Web Services do not support the notion of remote references that can be passed as arguments of operations. This already hints at a major difference in the way CORBA and Web Services should be used: CORBA is better for stateful servers; Web Services are better suited for stateless, message-oriented services.

**Binding.** The binding in a WSDL document indicates the type of transport that a Web Service uses. For example, if an XML document transmits its contents in the body, then the WSDL document needs to define that. If the document transmits its contents in Base64 encoding, then that would need to be defined here as well.

The binding name can be anything you wish. In this case, the name is GetStockQuoteBindingName.

Tools generating WSDL will use their own algorithms for naming the different sections.

The next element is soap:binding. It defines the style of the binding and the transport. There are two styles to choose from: RPC and document. RPC indicates that a message contains parameters and that there are return values. When the style attribute contains document, the request and response are passing XML documents within the body of the *SOAP* message.

The namespace in the transport indicates which protocol is used for the Web Service. In this case, we use HTTP by looking at the namespace. This is the required namespace for the HTTP transport. If using another transport such as SMTP, you can specify your own namespace, but the name must contain the transport like this: <http://advocatemedia.com/smtp/>. It is interesting to note that a WSDL document must specify the transport, but it cannot specify the address of the service according to the standard.

The next part of the WSDL document, where the input and output elements reside, defines the contents of the request and response messages. In this case, the example uses the contents of the document without any encoding. If either message contained any encoding such as Base64 or image/gif, it would be indicated here.

```
<binding name="GetStockQuoteBindingName" type="GetStockQuotePort ">
```

```

<soap:binding
    style="rpc"
    transport=" http://schemas.xmlsoap.org/soap/http"/>
<operation name="GetStockQuote">
    <soap:operation
        soapAction="http://advocatemedia.com/GetStockQuote"/>
    <input>
        <soap:body use="literal"/>
    </input>
    <output>
        <soap:body use="literal"/>
    </output>
</operation>
</binding>

```

The abstract definition of an interface does not describe how the interface is represented. This is the purpose of the binding tag. A binding describes how abstract definitions of a portType are converted into a concrete representation. This concrete representation is a combination of data formats and protocol. The following XML excerpt specifies that SOAP encoding is to be used for the operation deposit. As will be seen in the following section, SOAP defines how messages look on the network.

```

<binding name="AccountIFBinding" type="AccountIF">
    <operation name="deposit">
        <input>
            <body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                use="encoded"/>
        </input>
        <output>
            <body encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                use="encoded"/>
        </output>
    </operation>
    <!--similar bindings for withdraw and balance -->
    <binding transport="http://schemas.xmlsoap.org/soap/http" style="rpc"/>
</binding>

```

**Defining the service.** The only part of the Web Service left to define in the WSDL is the actual service. Now that the transport, actions, content, style of message, and many other things have been defined, it is time to actually define the service. The service element, as shown in the following code example, actually defines the name of the service along with documentation and the location of the service. The following is the code.

```

<service name="AdvocateMediaGetStockQuotes">
    <documentation>Simple Web Service to
                    Retrieve a stock quote</documentation>
    <port name="GetStockQuotePort"
        binding="myns:GetStockQuoteBindingName">
        <soap:address
            location="http://advocatemedia.com/GetStockQuote"/>
    </port>
</service>

```

The documentation element gives a developer the opportunity to provide some added information about what the Web Service accomplishes. The soap:address names the Web Service as a whole.

The last important XML tag of a WSDL specification is the service definition. It simply is a collection of ports, detailing the location of the Web Service. The following XML excerpt specifies that the MyAccountService of type AccountIFPort with binding AccountIFBinding can be accessed at the URL mentioned in the address tag.

```

<service name="MyAccountService">
    <port name="AccountIFPort" binding="AccountIFBinding">
        <address location="http://localhost:8080/account"/>

```

```

        </port>
    </service>
</definitions>
```

**The complete WSDL file.** Now that you have examined each section of a WSDL document, you need to look at it as a whole. Remember that WSDL is not necessarily for a person to read; it's more for an application to read in order to use the Web Service. Therefore, the XML might look pretty ugly in the complete document.

```

<definitions name="GetStockQuote"
    targetNamespace="http://advocatemedia.com/GetStockQuote.wsdl"
    xmlns:myns = "http://advocatemedia.com/GetStockQuote.wsdl"
    xmlns:myXsd = "http://advocatemedia.com/GetStockQuote.xsd"
    xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap"
    xmlns="http://schemas.xmlsoap.org">
    <types>
        <schema
            targetNamespace="http://advocatemedia.com/GetStockQuote.xsd"
            xmlns="http://www.w3.org/2000/10/XMLSchema">
            <element name="StockQuoteRequest">
                <complexType>
                    <all>
                        <element name="symbol" type="string"/>
                    </all>
                </complexType>
            </element>
            <element name="StockQuoteResponse">
                <complexType>
                    <all>
                        <element name="price" type="float"/>
                    </all>
                </complexType>
            </element>
        </schema>
    </types>
    <message name="GetStockQuote">
        <part name="body" element="myXSD:StockQuoteRequest"/>
    </message>
    <message name="GetStockQuoteResponse">
        <part name="body" element="myXSD:StockQuoteResponse"/>
    </message>
    <portType name="GetStockQuotePort">
        <operation name="GetStockQuote">
            <input message="myns:GetStockQuoteRequest"/>
            <output message="myns:GetStockQuoteResponse"/>
        </operation>
    </portType>
    <binding name="GetStockQuoteBindingName"
        type="StockQuoteBinding">
        <soap:binding
            style="rpc"
            transport=" http://schemas.xmlsoap.org/soap/http"/>
        <operation name="GetStockQuote">
            <soap:operation
                soapAction="http://advocatemedia.com/GetStockQuote"/>
            <input>
                <soap:body use="literal"/>
            </input>
            <output>
                <soap:body use="literal"/>
            </output>
        </operation>
    </binding>
    <service name="AdvocateMediaGetStockQuotes">
        <documentation>Simple Web Service to
            Retrieve a stock quote</documentation>
```

```

<port name="GetStockQuotePort"
      binding="myns:GetStockQuoteBindingName">
    <soap:address
      location="http://advocatemedia.com/GetStockQuote"/>
  </port>
</service>
</definitions>

```

**Using import.** There is an alternate and perhaps easier way of writing the WSDL file. This involves defining all the types in an *XML Schema Reduced* (XSD) file while putting all the other definitions relevant to the Web Service in the WSDL file. This way, the schema element and all of its children are in a separate file. If you are using the same elements in different Web Services, you can easily move the schema definitions from application to application. For example, there may be several stock-related Web Services that use the same types and variables. This way one XSD file could support all the different services.

Here is the GetStockQuote.xsd example.

```

<schema
  targetNamespace="http://advocatemedia.com/GetStockQuote.xsd"
  xmlns="http://www.w3.org/2000/10/XMLSchema">
  <element name="StockQuoteRequest">
    <complexType>
      <all>
        <element name="symbol" type="string"/>
      </all>
    </complexType>
  </element>
  <element name="StockQuoteResponse">
    <complexType>
      <all>
        <element name="price" type="float"/>
      </all>
    </complexType>
  </element>
</schema>

```

From within the WSDL file, the import element is used to bring in the element definitions in GetStockQuote.xsd, as shown in the following code.

```

<definitions name="GetStockQuote"
  targetNamespace="http://advocatemedia.com/GetStockQuote.wsdl"
  xmlns:myns = "http://advocatemedia.com/GetStockQuote.wsdl"
  xmlns:myXsd = "http://advocatemedia.com/GetStockQuote.xsd"
  xmlns:soap = "http://schemas.xmlsoap.org/wsdl/soap"
  xmlns="http://schemas.xmlsoap.org">
  <import namespace="http://advocatemedia.com/GetStockQuote.xsd"
    location ="http://advocatemedia.com/GetStockQuote.xsd">
  <message name="GetStockQuote">
    <part name="body" element="myXSD:StockQuoteRequest"/>
  </message>
  <message name="GetStockQuoteResponse">
    <part name="body" element="myXSD:StockQuoteResponse"/>
  </message>
  <portType name="GetStockQuotePort">
    <operation name="GetStockQuote">
      <input message="myns:GetStockQuoteRequest"/>
      <output message="myns:GetStockQuoteResponse"/>
    </operation>
  </portType>
  <binding name="GetStockQuoteBindingName"
    type="StockQuoteBinding">
    <soap:binding
      style="rpc"
      transport=" http://schemas.xmlsoap.org/soap/http"/>

```

```

<operation name="GetStockQuote">
    <soap:operation
        soapAction="http://advocatemedia.com/GetStockQuote"/>
    <input>
        <soap:body use="literal"/>
    </input>
    <output>
        <soap:body use="literal"/>
    </output>
</operation>
</binding>
<service name="AdvocateMediaGetStockQuotes">
    <documentation>Simple Web Service to
        Retrieve a stock quote</documentation>
    <port name="GetStockQuotePort"
        binding="myns:GetStockQuoteBindingName">
        <soap:address
            location="http://advocatemedia.com/GetStockQuote"/>
    </port>
</service>
</definitions>

```

Notice that only the difference between this and the original complete WSDL example is the import element.

The Web Service received several requests for quotes and then returned several prices. For the sake of simplifying the WSDL code, the *SOAP* code simply receives one stock symbol and then transmits one price.

In the following code, you'll see that the single symbol "C" (for Citicorp) resides in the code.

```

POST /stockquotes HTTP/1.1
Host: www.advocatemedia.com:80
Content-Type: text/xml; charset=utf-8
Content-Length: 482
SOAPAction: "http://www.advocatemedia.com/webservices/getquote"

<?xml version='1.0' ?>
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/soap-envelope"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">
<env:Body>
    <StockQuoteRequest xmlns="http://advocatemedia.com/Examples">
        <symbol xsi:type="string">
            C
        </symbol>
    </StockQuoteRequest>
</env:Body>
</env:Envelope>

```

The response sends one single price back to the client. In the following example, you'll see that the service sends the price of the stock back to the client in the form of float.

```

HTTP/1.1 200 OK
Connection: close
Content-Length: 659
Content-Type: text/xml; charset=utf-8

<?xml version='1.0' ?>
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/soap-envelope"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/">

```

```

<env:Body>
  <StockQuoteResponse xmlns="http://advocatemedia.com/Examples"
    <price xsi:type="float">
      53.21
    </price>
  </StockQuoteResponse>
</env:Body>
</env:Envelope>

```

Code Example 3.29 shows a WSDL document for a weather Web service that returns a given city's weather information. The Web service, which uses SOAP as the communication protocol, expects to receive the city name as String type data and sends String type data as its response.

#### **Example 3.29. WSDL Document for Weather Web Service**

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WeatherWebService"
  targetNamespace="urn:WeatherWebService"
  xmlns:tns="urn:WeatherWebService"
  xmlns="http://schemas.xmlsoap.org/wsdl/"
  xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/">
<types/>
<message name="WeatherService_getWeather">
  <part name="String_1" type="xsd:string"/>
</message>
<message name="WeatherService_getWeatherResponse">
  <part name="result" type="xsd:string"/>
</message>
<portType name="WeatherService">
  <operation name="getWeather" parameterOrder="String_1">
    <input message="tns:WeatherService_getWeather"/>
    <output
      message="tns:WeatherService_getWeatherResponse"/>
  </operation>
</portType>
<binding name="WeatherServiceBinding" type="tns:WeatherService">
  <operation name="getWeather">
    <input>
      <soap:body use="literal"
        namespace="urn:WeatherWebService"/>
    </input>
    <output>
      <soap:body use="literal"
        namespace="urn:WeatherWebService"/>
    </output>
    <soap:operation soapAction="" /></operation>
  <soap:binding
    transport="http://schemas.xmlsoap.org/soap/http"
    style="rpc"/>
</binding>
<service name="WeatherWebService">
  <port name="WeatherServicePort"
    binding="tns:WeatherServiceBinding">
    <soap:address
      location="http://mycompany.com/weatherservice"/>
  </port>
</service>
</definitions>

```

Example 3.30 is the portion of the document the Amazon Web Services WSDL document.

#### **Example 3.30. The AWS WSDL document (portions of it)**

```
<wsdl:definitions xmlns:typens="http://soap.amazon.com"
```

```

xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:soap="http://schemas.xmlsoap.org/wsdl/soap/"
xmlns:soapenc="http://schemas.xmlsoap.org/soap/encoding/"
xmlns:wsdl="http://schemas.xmlsoap.org/wsdl/"
xmlns="http://schemas.xmlsoap.org/wsdl/"
targetNamespace="http://soap.amazon.com" name="AmazonSearch">
<wsdl:types>
    <xsd:schema xmlns="" xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        targetNamespace="http://soap.amazon.com">
        <xsd:complexType name="ProductLineArray">
            <xsd:complexContent>
                <xsd:restriction base="soapenc:Array">
                    <xsd:attribute ref="soapenc:arrayType"
                        wsdl:arrayType="typens:ProductLine[]"/>
                </xsd:restriction>
            </xsd:complexContent>
        </xsd:complexType>
        <xsd:complexType name="ProductLine">
            <xsd:all>
                <xsd:element name="Mode" type="xsd:string" minOccurs="0"/>
                <xsd:element name="ProductInfo" type="typens:ProductInfo"
                    minOccurs="0"/>
            </xsd:all>
        </xsd:complexType>
        <xsd:complexType name="ProductInfo">
            <xsd:all>
                <xsd:element name="TotalResults" type="xsd:string"
                    minOccurs="0"/>
                <xsd:element name="TotalPages" type="xsd:string"
                    minOccurs="0"/>
                <xsd:element name="ListName" type="xsd:string"
                    minOccurs="0"/>
                <xsd:element name="Details" type="typens:DetailsArray"
                    minOccurs="0"/>
            </xsd:all>
        </xsd:complexType>
        <xsd:complexType name="DetailsArray">
            <xsd:complexContent>
                <xsd:restriction base="soapenc:Array">
                    <xsd:attribute ref="soapenc:arrayType"
                        wsdl:arrayType="typens:Details[]"/>
                </xsd:restriction>
            </xsd:complexContent>
        </xsd:complexType>
        <xsd:complexType name="Details">
            <xsd:all>
                <xsd:element name="Url" type="xsd:string" minOccurs="0"/>
                <xsd:element name="Asin" type="xsd:string" minOccurs="0"/>
                <xsd:element name="ProductName" type="xsd:string"
                    minOccurs="0"/>
                <xsd:element name="Catalog" type="xsd:string"
                    minOccurs="0"/>
                <!-- Edited for length -->
                <xsd:element name="Authors" type="typens:AuthorArray"
                    minOccurs="0"/>
                <xsd:element name="ListPrice" type="xsd:string"
                    minOccurs="0"/>
                <xsd:element name="OurPrice" type="xsd:string"
                    minOccurs="0"/>
                <xsd:element name="UsedPrice" type="xsd:string"
                    minOccurs="0"/>
                <xsd:element name="NumberOfPages" type="xsd:string"
                    minOccurs="0"/>
            </xsd:all>
        </xsd:complexType>
        <xsd:complexType name="AuthorArray">

```

```

<xsd:complexContent>
    <xsd:restriction base="soapenc:Array">
        <xsd:attribute ref="soapenc:arrayType"
                      wsdl:arrayType="xsd:string[]"/>
    </xsd:restriction>
</xsd:complexContent>
</xsd:complexType>
<xsd:complexType name="AsinRequest">
    <xsd:all>
        <xsd:element name="asin" type="xsd:string"/>
        <xsd:element name="tag" type="xsd:string"/>
        <xsd:element name="type" type="xsd:string"/>
        <xsd:element name="devtag" type="xsd:string"/>
        <xsd:element name="offer" type="xsd:string"
                      minOccurs="0"/>
        <xsd:element name="offerpage" type="xsd:string"
                      minOccurs="0"/>
        <xsd:element name="locale" type="xsd:string"
                      minOccurs="0"/>
    </xsd:all>
</xsd:complexType>
</xsd:schema>
</wsdl:types>
<message name="AsinSearchRequest">
    <part name="AsinSearchRequest" type="typens:AsinRequest"/>
</message>
<message name="AsinSearchResponse">
    <part name="return" type="typens:ProductInfo"/>
</message>
<portType name="AmazonSearchPort">
    <operation name="AsinSearchRequest">
        <input message="typens:AsinSearchRequest"/>
        <output message="typens:AsinSearchResponse"/>
    </operation>
</portType>
<binding name="AmazonSearchBinding" type="typens:AmazonSearchPort">
    <soap:binding style="rpc"
                  transport="http://schemas.xmlsoap.org/soap/http"/>
    <operation name="AsinSearchRequest">
        <soap:operation soapAction="http://soap.amazon.com"/>
        <input>
            <soap:body use="encoded"
                       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                       namespace="http://soap.amazon.com"/>
        </input>
        <output>
            <soap:body use="encoded"
                       encodingStyle="http://schemas.xmlsoap.org/soap/encoding/"
                       namespace="http://soap.amazon.com"/>
        </output>
    </operation>
</binding>
<service name="AmazonSearchService">
    <port name="AmazonSearchPort" binding="typens:AmazonSearchBinding">
        <soap:address location="http://soap.amazon.com/onca/soap2"/>
    </port>
</service>
</wsdl:definitions>

```

This document defines everything about AWS in terms of the ASIN search; it contains definitions for all of the other search capabilities that Amazon provides.

### **3.3.4 SOAP**

SOAP enables objects not known to one another to communicate; that is, to exchange messages. SOAP, a wire protocol similar to Internet Inter-ORB Protocol (IIOP) and Java Remote Method Protocol (JRMP), is a text-based protocol that uses an XML-based data encoding format and HTTP/SMTP to transport messages. SOAP is independent of both the programming language and the operational platform, and it does not require any specific technology at its endpoints, making it completely agnostic to vendors, platforms, and technologies. Its text format also makes SOAP a firewall-friendly protocol. Moreover, SOAP is backed by leading industrial players and can be expected to have universal support.

SOAP is a relatively simple and straightforward protocol that has been developed as a W3C recommendation. The latest version is the SOAP Version 1.2 Recommendation from 2003.

With the emergence of Web services, SOAP has become the *de facto* communication protocol standard for creating and invoking applications exposed over a network. SOAP is similar to traditional binary protocols like IIOP (CORBA) or JRMP (RMI), but instead of using a binary data representation, it adopts text-based data representation using XML.

Using XML notation, SOAP defines a lightweight wire protocol and encoding format to represent data types, programming languages, and databases. SOAP can use a variety of Internet standard protocols (such as HTTP and SMTP) as its message transport, and it provides conventions for representing communication models like remote procedural calls (RPCs) and document-driven messaging. This enables inter-application communication in a distributed environment and interoperability between heterogeneous applications over the networks. With its widespread acceptance by leading IT vendors and Web developers, SOAP is gaining popularity and adoption in most popular business applications for enabling them as Web services. It is important to note that SOAP is an ongoing W3C effort in which leading IT vendors are participating in order to come to a consensus on such important tasks associated with XML-based protocols and to define their key requirements and usage scenarios.

#### **3.3.4.1 SOAP Concept**

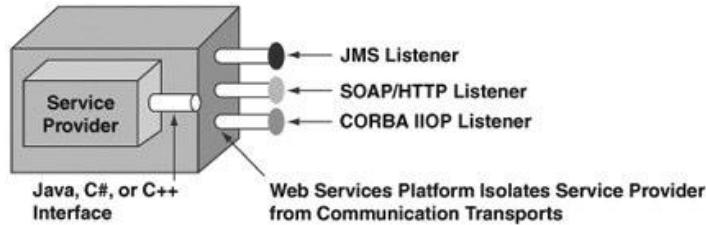
SOAP, now an empty acronym, used to be the Simple Object Access Protocol, and sometimes is expanded as Service-Oriented Access Protocol. SOAP is an XML-based protocol for passing information back and forth over a network. Defined using XML, SOAP is a very flexible protocol that does not rely on a single language to produce or use it. This flexibility, in turn, allows programs written in different languages on different operating systems to still communicate effectively. A typical example, especially with web services, is a web application written in ASP.NET on a Windows 2003 Server communicating with a web service written in Perl on an Ubuntu server.

SOAP defines the structure of an XML document, rules, and mechanisms that can be used to enable communication between applications. It does not mandate a single programming language or a platform, nor does it define its own language or platform.

Web services are based on SOAP, and SOAP is transport-neutral. Although SOAP defines only one mandatory transport binding (HTTP) and the WS-I basic profile only requires SOAP over HTTP, other bindings can be defined for other transports. For many applications, HTTP is perfectly acceptable; however, HTTP deliberately omits certain features (such as persistent sessions) so that it can scale up and work well over the Web. However, many organizations want to use Web services on their internal networks where (a) the scalability issues of the World Wide Web are not present and (b) they have already invested heavily in designing and deploying an enterprise messaging infrastructure that is based on JMS, WebSphere MQ, CORBA Notification, or Tibco Rendezvous. These organizations want to take advantage of SOAP and WSDL while also taking advantage of their existing enterprise messaging infrastructure.

One of the advantages of a well-defined service contracts is that the logical contract can be defined in a manner that is independent of the underlying communication transport or middleware. This allows the service requesters and service providers to be developed without being tied to a particular transport. WSDL supports this approach by providing extension mechanisms so that alternative (non-HTTP) communication transports can be specified. This makes it possible to isolate the application-level code from the underlying communication transports:

1. The application can take advantage of the qualities of service provided by the enterprise messaging infrastructure (such as guaranteed message delivery and security) without being tightly bound to any particular protocol or middleware.
2. The underlying communication transport or middleware can be changed at any time by simply modifying the WSDL without affecting the application code at all.
3. A service provider can be available over multiple communication transports (such as HTTP, HTTPS, JMS, SMTP, and FTP) simultaneously, and service requesters can choose which one to use depending on the qualities of service they require (see Figure 3.23).



**Figure 3.23. Defining multiple transports for a service.**

The specification that really kick-started the Web services revolution was Simple Object Access Protocol (SOAP) 1.1, which defined an XML envelope for Web service messages, a processing model for Web service intermediaries, and an encoding algorithm for serializing objects as XML. The SOAP envelope was extremely simple, consisting of a *body* and an optional *header*. The SOAP body contained the application payload, and the SOAP header contained any other non-application data such as security, reliability, or transaction information. The separation of messages into a header and a body is a well-accepted design practice. The SOAP processing model specified how network intermediaries could process the SOAP header information before delivering the SOAP body to the Web service or client.

The SOAP envelope and processing model were fairly uncontroversial and relatively easy to implement correctly. On the other hand, the SOAP encoding algorithm proved to be much more problematic. The root cause of the difficulty was that there is no universally accepted definition of objects. Each object-oriented programming language implements many common features of objects, but adds differences. For example, C++ supports multiple inheritance but Java only supports single inheritance. There is simply no way to faithfully interchange objects between arbitrary programming languages.

But even if there was a commonly accepted way to exchange objects, that would still be the wrong way to build robust distributed systems. If you look at a typical programming language object, it contains more than just state information. It also contains fields used to make navigation and other operations more efficient. For example, a linked list really just represents a sequence of objects, but it contains forward and backward pointers. There is no purpose in serializing these redundant fields in a Web service message. They increase the bulk of the message, and the receiving end may elect to represent the sequence in some other way, for example, as an array. Objects are wonderful for implementing applications but are really not a good basis for designing Web service interfaces. Furthermore, a Web service should support a wide variety of application types, not just object-oriented systems.

As the name SOAP suggests, SOAP encoding was motivated by a desire to create a distributed object technology for the Web. Earlier distributed object technologies such as CORBA, Java RMI, and DCOM failed to gain significant traction on the Internet. When XML emerged, Microsoft proposed that it could be used as an architecturally neutral way to serialize graphs of objects. SOAP was proposed as the carrier for these serialized object graphs and the serialization algorithm was dubbed *SOAP encoding*. The fact that XML was used as the serialization syntax was incidental. To use SOAP encoding, you had to have matching client and service implementations that understood the SOAP encoding algorithm. The client and the server were assumed to both be implemented in conventional object-oriented programming languages. The flaw in this approach is that exchanging objects is really not what the Web is all about. The Web is highly heterogeneous, and there are many types of clients and ways of processing XML. For example, XML can be processed using DOM, SAX, StAX, XPath, XSLT, and XQuery to name a few. In fact, one of the design principles behind XML is that it should be easily processable by a variety of applications. SOAP encoding clearly violates that principle.

A better approach to the design of Web service interfaces is to view Web service operations as document exchanges. After all, business in the real world is transacted by the exchange of documents. For example, I fill out a driver's license application form and the motor vehicle department sends me my driver's license. I do not

remotely invoke the driver's license procedure or send the motor vehicle department a serialized driver's license application form object graph. Documents are very natural, and XML is an excellent way to represent them in information systems. XML Schema is the W3C standard type system for XML documents. But XML is really just a representation of a document, albeit a very convenient one for many purposes. In general, there may be other useful representations of documents. For example, if one just want to display the document to a human, then HTML or PDF is a better representation. On the other hand, if one want to use the document in a Service-Oriented Architecture (SOA) application, then document/literal SOAP is probably the best representation.

### 3.3.4.2 SOAP Vocabulary

The *SOAP* standard dictates how the XML looks within the *SOAP* document, how the contents of that message are transmitted, and how that message is handled at both the sender and receiver. *SOAP* also provides a standard set of vocabulary. This chapter introduces that vocabulary, shows different applications of that vocabulary, and shows what the XML in a *SOAP* document looks like.

As with any technology, *SOAP* comes with its own set of vocabulary. There are several terms used frequently to describe different aspects of the *SOAP* standard. You'll find that many developers use these terms without truly understanding their meaning. So it's important to take the time to understand what each term means and how it applies to both the *SOAP* standard and to an actual Web Service.

The SOAP standard is not just an XML standard. The standard includes how SOAP messages should behave, the different transports used, how errors get handled, and more.

The terminology related to *SOAP* comes in two different categories: **transmission** and **message**. The terms related to transmission deal with describing how *SOAP* messages relate to the protocol, how the different *SOAP* nodes converse, and so on. On the other hand, terms related to the XML within *SOAP* fall into the message category.

The *SOAP* standard also defines a small set of XML elements to encapsulate the data transmitted between nodes. There are really only a few elements because the body of the message can vary depending on the implementation. This flexibility is allowed by the standard.

As this standard moves forward, the name will change to XMLP for the XML Protocol. *SOAP*'s evolution has been muddied by the fact that it came from Microsoft. XMLP is a complete rewrite of the standard so that Web Services become even more cross-platform compatible. Because of this change, much of this vocabulary must evolve. Although the names will probably change, the general idea of each term will remain the same.

**Binding.** This describes how a *SOAP* message works with a transport protocol such as HTTP, SMTP, or FTP to move across the Internet. It is important that *SOAP* moves across a standard protocol in order to communicate with other Web Service products. Before *SOAP*, many developers created their own method of transmitting XML documents through a network. This works fine as long as the transmission is limited within a particular team. If, however, you need to work with another group either within or outside your company this becomes difficult because of training and possible modification to work with an XML transmission they may be using. By using a standard XML document on standard protocols, the work needed for collaboration will be minimal.

**Message Exchange Pattern (MEP).** This describes how a *SOAP* document gets exchanged between a client and server. The *SOAP* message possesses a binding, such as HTTP, so that it can move across the Internet. The conversation between the client and server, both known as nodes, determines what actions both take. Remember that *SOAP* is an XML encapsulation of RPC. Therefore, the MEP is completely request and response between the client and server (or other nodes). Thus, if there needs to be several interactions between nodes, this takes several requests and responses to complete transmission. This differs from other remote object technologies such as CORBA where the entire conversation occurs over one single connection.

**Application.** A *SOAP* application is simply an application that uses *SOAP* in some way. Some applications may be entirely based on the *SOAP* standard, such as the stock Web Services example shown later in the chapter, or may just use the *SOAP* standard to receive code or software updates. Remember an application can produce, consume, or be an intermediary (or router).

**Node.** A node's responsibility can include sending, receiving, processing, or retransmitting a *SOAP* message. A node is just a piece of software that properly handles a *SOAP* document dependent on its role. Besides transmission, a node is also responsible for enforcing that the XML contained in the *SOAP* document is grammatically correct according to the *SOAP* standard.

**Role.** A *SOAP* role defines what a particular node does. It may be a sender, receiver, or intermediary.

**Sender.** The node sending the *SOAP* request is the *SOAP* sender. If you think of a client/server example, when a client first makes a request, it sends a message to the server asking for some information. In the upcoming Stock Quote Example, the client sends a request to the Stock Quote Server. In this case, the client acts as the *SOAP* sender by transmitting a message asking for a quote.

**Receiver.** A server that receives the *SOAP* request is obviously the receiver. This is the server in the client/server model. The Stock Quote Example later on in the chapter has a server that receives the request for stock quotes and then returns the appropriate values.

**Intermediary.** An intermediary looks at a *SOAP* message, perhaps acts on some of the information in the message, and then looks at the *SOAP* document for more information on where to pass the information in the document next. A *SOAP* intermediary essentially acts like a router in a network. A router takes a look at a packet of information moving through a network, finds the packet's next destination, and then sends it to that destination. A *SOAP* intermediary does the same thing but it's looking at *SOAP* messages and information in the XML to send the message to the proper location. This occurs when a large corporation possesses many *SOAP* servers that perform different functions, and the *SOAP* intermediary may have access to the firewall. Once it receives the information, it looks at the XML to see where to send the message next. It may act on or modify the data before this retransmission, but it is not necessary.

**Message Path.** A *SOAP* message moves from sender to receiver perhaps through several intermediaries. The resulting route the message takes is the Message Path.

**Initial Sender.** The node sending the first *SOAP* request is the initial *SOAP* sender.

**Feature.** A *SOAP* feature is a piece of functionality in software supporting *SOAP* that deals with a feature of *SOAP*. Examples include securing the transaction with a secure protocol or the software acting as an intermediary.

**Message.** This is the XML document transmitted by either a *SOAP* sender or receiver. A sender or client creates an XML document containing the information the client needs from the server. Once that document is transmitted, the server parses the information in the document to access the various values and then creates a new *SOAP* message as the response.

**Envelope.** This is the root element of the *SOAP* XML document. The *SOAP* document contains several namespace definitions but the elements related to the *SOAP* message will have ENV: as the prefix. Examples later in the chapter describe the XML in greater detail.

**Header.** The first part of a *SOAP* message contains a header block in XML that is for the routing and processing of the *SOAP* message. This data is separate from the body of the document which has information related to the object call being made.

**Header Block.** A *SOAP* header containing several delimited sections or blocks of information has a header block. These header blocks come with processes that include *SOAP* intermediaries because a node needs to know where to send the message to next.

**Body.** The body of the message actually contains the information for the object to process the information. The body is still in XML and, once parsed, the information goes to the object. The object processes the information and the result is put into the *SOAP* body of the returned document.

**Fault.** This is simply a piece of information in the XML of a *SOAP* document containing information related to any error that may have occurred at one of the *SOAP* nodes.

**Quick Reference.** Table 3.8 provides a summary of all connection-related terms. Table 3.9 summarizes *SOAP*-related terms.

**Table 3.8: Provides a Quick Summary of All the Connection-Related Terms**

Connection Term	Description
SOAP	Standard defining how the XML document looks and how the information moves across the Internet.
SOAP Binding	Describes how SOAP interacts with a standard transport protocol such as HTTP or SMTP.
SOAP message Exchange Pattern (MEP)	This is the conversation a client and server has while exchanging information.
SOAP Application	An application that consumes or creates SOAP messages.
SOAP Node	A server that somehow interacts with the SOAP message Exchange Pattern.

**Table 3.8: Provides a Quick Summary of All the Connection-Related Terms**

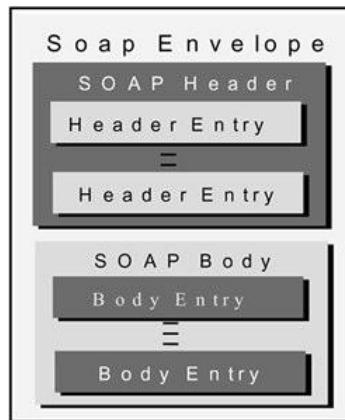
Connection Term	Description
SOAP Role	A SOAP node may have one of three roles or jobs: sender, receiver, and intermediary.
SOAP Sender	A SOAP sender is the node sending a request to another node.
SOAP Receiver	A SOAP receiver is the final processor or destination of the request.
SOAP Intermediary	Acts as a router or relay by passing the SOAP message onto the next node. Relies on information in the message to find the next node.
Message Path	The path or route the SOAP message follows during processing.
Initial SOAP Sender	The node that originated the SOAP request.
SOAP Feature	Relates to pieces of the SOAP standard such as security or errors.

**Table 3.9: A Summary of All the Terms Related to the SOAP message**

SOAP message Term	Description
SOAP message	The XML document transmitted between SOAP nodes.
SOAP Envelope	The root element of the SOAP XML document.
SOAP Header	The top portion of the SOAP XML document that contains information relevant to the processing of the message.
SOAP Header Block	If a header contains a lot of information, several section or blocks occur in the header to separate the information.
SOAP Body	The part of the SOAP XML document right below the header that contains information for the actual object call.
SOAP Fault	Information in the SOAP document relevant to any error that occurred.

### 3.3.4.3 A Short Description

Certain elements are required to make up a proper SOAP document: **an envelope and a body**. In addition, there are optional elements: **a header and a fault**. All of these elements are declared in the default namespace for SOAP, while the data types and element encoding are contained in their own namespace. SOAP defines an envelope, which contains a SOAP body, within which the message is included, and an optional SOAP-specific header. The whole envelope—body plus header—is one complete XML document. (See Figure 3.24)



**Fig. 3.24. SOAP Message Structure**

The header entries may contain information of use to recipients, and these header entries may also be of use to intermediate processors since they enable advanced features. The body, which contains the message contents, is consumed by the recipient. SOAP is agnostic about the message contents; the only restriction is that the message be in XML format.

When creating a new SOAP document, you must remember the following syntax rules to ensure that the document is structured properly:

- A SOAP message must be an XML encoded document.
- A DTD reference must not be included in a SOAP document.
- XML processing instructions must not be included in a SOAP document.

- The SOAP Envelope namespace must be used in the document.
- The SOAP Encoding namespace must be used in the document.

Following these syntax rules, the basic skeleton for SOAP looks like this:

```
<?xml version="1.0" encoding="utf-8"?>
<soap:Envelope xmlns:soap="http://www.w3.org/2001/12/soap-envelope"
    soap:encodingStyle="http://www.w3.org/2001/12/soap-encoding">
    <soap:Header>
        <!-- Header information -->
    </soap:Header>
    <soap:Body>
        <!-- Body Information -->
    </soap:Body>
</soap:Envelope>
```

Example 3.31 shows what a request may look like using Amazon Web Services (AWS) to get details regarding a book. The Amazon Standard Item Number (ASIN) is how Amazon tracks every item that it sells. In the case of books, the ASIN is the same as the book's ISBN.

#### **Example 3.31. A SOAP request using AWS**

```
<?xml version="1.0" encoding="utf-8" ?>
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Body>
    <nsp1:AsinSearchRequest xmlns:nsp1="urn:PI/DevCentral/SoapService">
        <AsinSearchRequest xsi:type="m:AsinRequest">
            <asin>0596528388</asin>
            <page>1</page>
            <mode>books</mode>
            <tag>associate tag</tag>
            <type>lite</type>
            <dev-tag>developer token</dev-tag>
            <format>xml</format>
            <version>1.0</version>
        </AsinSearchRequest>
    </nsp1:AsinSearchRequest>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Once Amazon receives this request, its service will process it and return an XML response to the client. In this request, the response is an XML document specific to Amazon's products, as was requested in the SOAP request. Many times, a SOAP request will be answered with a SOAP response because there is no choice for a response format. It is important to know what you will be receiving from the web service when you make a request!

**Interoperability through SOAP.** Web Services realize interoperability through SOAP. Interoperability defines the “language” that different Web Service implementations use to exchange messages. As already mentioned, Web Services use XML for this job as well. The *SOAP* is *based on* content of the messages flowing between client and server are marked up via XML. Special tags are introduced for the purpose of marshalling actual parameters of remote operations.

In the following we present two SOAP messages: one request message and one response message. Just like with GIOP, the request message is sent from client to server:

```
<Envelope>
    <Body>
        <deposit>
            <amount type="int">700</amount>
        </deposit>
```

```

</Body>
</Envelope>

```

The above XML has been simplified for the purpose of this example. The request is typically transported via HTTP from client to server. The tag Envelope frames the whole SOAP request message. It contains a body denoted by the XML tag with the same name. The operation is encoded as the content of the body tag. The operation name is represented by its own tag, as are the actual parameters that accompany the invocation. Note that the actual parameters are *SOAP request* accompanied by type information.

The following XML shows a SOAP response:

```

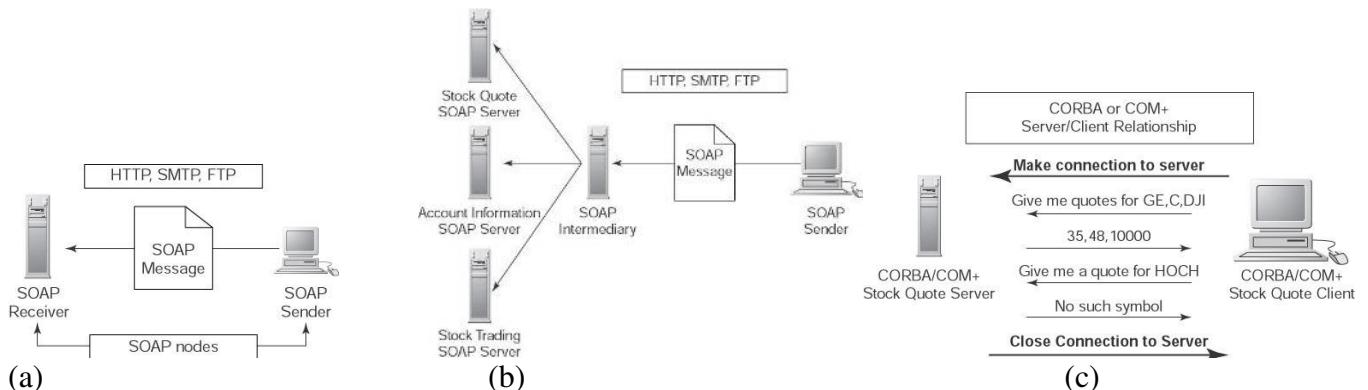
<Envelope>
  <Body>
    <depositResponse/>
  </Body>
</Envelope>

```

This PDU will be sent by the server to the client in response to a request. Once again, the whole message is framed by the Envelope tag. This time, the body contains all result parameters that accompany the response. Note that there is no special message ID.

SOAP is a text-based protocol. While the proponents of text-based protocols argue that it is a nice feature to actually see what is sent between client and server, there are also some serious drawbacks. First of all, there should not be any need to watch the wire protocol. Debugging happens on the application level, and there is no need to inspect the content of PDUs. Second, text-based protocols incur a high runtime overhead. They use up more bandwidth, but more importantly the stubs and skeletons have to handle XML messages. Having to parse and construct XML messages can only be done at high costs compared to a binary protocol. Therefore SOAP does not seem a good candidate for applications with high-frequency transactions.

**A Simple SOAP Transaction.** Figure 3.25(a) shows a simple model of a *SOAP* transaction. The first item to note is the different protocols available for transmission. FTP, HTTP, and SMTP are available. This provides a great deal of flexibility especially if consumers of your Web Service are in an area that doesn't have a good connection to the Web but still has e-mail access.



**Figure 3.25:** (a) This image demonstrates a simple SOAP transaction. (b) How a SOAP message can be routed by a SOAP intermediary (c) How two SOAP nodes converse over a stock quote

The *SOAP* message is the XML document being sent to the *SOAP* receiver. The message contains information in XML that is eventually processed by an object on the receiver. The *SOAP* nodes are both the *SOAP* sender and receiver. Think of the *SOAP* sender as a Web page or an application that consumes the Web Service, and the *SOAP* receiver as the server that hosts the Web Service.

**Using an Intermediary.** In the next example, Figure 3.25(b), a *SOAP* intermediary node is introduced. As mentioned earlier, an intermediary acts as a Web Services router sending requests to the appropriate location based on data found in the XML of the *SOAP* document.

In this case, the *SOAP* sender is a client for a large brokerage house. This brokerage house possesses several *SOAP* receivers each with a different function. By having the *SOAP* intermediary, the brokerage house only needs to publish one address to fulfill all the different requests. Once a *SOAP* message reaches the intermediary, the XML inside the message reveals the next node the request needs to go to. In this case, the brokerage house is offering services for stock quotes, account information, and stock trading. Thus, the XML contains information telling the intermediary where to send the message next. The important thing to note is that the intermediary can act on other information in the *SOAP* message, but it is not required to do so.

**How SOAP Differs.** *SOAP* is just a request and response system. This is much like a browser making a request to a Web page. You enter the address into the browser and the request goes out to the Internet and finds the server. The server sends the response in the form of HTML; then there is no longer a connection between client & server.

Figure 3.25(c) shows this request and response pattern between two *SOAP* nodes. This is a conversation between the two nodes about getting some stock quotes. In the first request, the *SOAP* client asks for stock quotes for the symbols: “GE,” “C,” and “DJI.” The response is “35,” “48,” and “10,000.” The *SOAP* client then asks for a quote for the symbol “HOCH,” and the receiver sends back information stating there is not such symbol.

The difference between a client and server in COM+, CORBA, or another remote object technology is the fact that the entire conversation occurs over a single connection. Figure 3.25(c) shows the same exchange of information over stock quotes as the previous example. In this case, however, there is only one connection, and it is held until the entire conversation is complete. This connection is much more like telnet—a session is held constant between a client and server. Because the connection is constant, the response to multiple requests will most likely be faster than a request and response model such as Web Services, but the implementation is far more difficult.

### 3.3.4.4 Basic SOAP Document

*SOAP* uses XML to describe the data transmitted between nodes. The tags used are to describe the document, header, and body. The tags in the body are usually specific to a vendor’s implementation of *SOAP* because the standard doesn’t dictate the XML appearing there. The standard does, however, dictate the information that appears in the header and the how the envelope, body, and header elements look.

Start off with a headerless *SOAP* document such as the following.

```
<?xml version='1.0' ?>
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/SOAP-envelope">
    <env:Body>
    </env:Body>
</env:Envelope>
```

This is a simple *SOAP* document that doesn’t say anything, but note the namespace definition and the fact that there is an envelope and a body. The document is just a container for an object to get data from, process, and then put back into the document to transmit it back to the client. Let’s consider the request for stock quote information again.

```
<?xml version='1.0' ?>
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/SOAP-envelope">
    <env:Body>
        <stockquote:symbolist
            xmlns:stockquote="http://advocatemedia.com/stocks">
            <stockquote:symbol>C</stockquote:symbol>
            <stockquote:symbol>GE</stockquote:symbol>
            <stockquote:symbol>DJI</stockquote:symbol>
        <stockquote:symbolist>
    </env:Body>
</env:Envelope>
```

Now we have information specific to the object that returns stock quotes. Note that the stockquote tags are not defined by the *SOAP* standard; rather, they are defined by the developer responsible for the software in which the Stock Quote Web Service was created.

Code Example 3.32 shows a simple but complete example of a *SOAP* request for obtaining a stock quote.

### **Example 3.32. Example *SOAP* Request**

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="SoapEnvelopeURI"
    SOAP-ENV:encodingStyle="SoapEncodingURI">
    <SOAP-ENV:Header>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <m:GetLastTradePrice xmlns:m="ServiceURI">
            <tickerSymbol>SUNW</tickerSymbol>
        </m:GetLastTradePrice>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

This example shows how a *SOAP* message is encoded using XML and illustrates some *SOAP* elements and attributes. All *SOAP* messages must have an *Envelope* element and must define two namespaces: One namespace connotes the *SOAP* envelope (`xmlns:SOAP-ENV`) and the other indicates the *SOAP* encoding (`SOAP-ENV:encodingStyle`). *SOAP* messages without proper namespace specification are considered invalid messages. The `encodingStyle` attribute is important, as it is used to specify serialization rules for the *SOAP* message. Moreover, there can be no DTD referrals from within *SOAP* messages.

While optional, the *Header* element when used should be the first immediate child after the *Envelope*. The *Header* element provides a way to extend the *SOAP* message by specifying additional information such as authentication and transactions. Specifying this additional information as part of the *Header* tells the message recipient how to handle the message.

There are many attributes that can be used in the *SOAP Header* element. For example, the `actor` attribute of the *Header* element enables a *SOAP* message to be passed through intermediate processes en route to its ultimate destination. When the `actor` attribute is absent, the recipient is the final destination of the *SOAP* message. Similarly, many other attributes may be used. However, this chapter does not address these details.

The *Body* element, which must be present in all *SOAP* messages, must follow immediately after the *Header* element, if it is present. Otherwise, the *Body* element must follow immediately after the start of the *Envelope* element. The *Body* contains the specification of the actual request (such as method calls). The *Fault* element in the *SOAP Body* enables error handling for message requests.

The header in a *SOAP* document is optional, but when it is present it can determine the next node of a *SOAP* request when sent through an intermediary. Back in Figure 3.25(b), a *SOAP* intermediary looks at the contents of the XML to determine where to route the file next.

Remember that a *SOAP* intermediary can act or modify the data in the document and then pass it to the next node, or just simply act as a router and pass it to the next appropriate node. For an intermediary to pass the information along, the document must contain the information to send it to the next node. The following *SOAP* example possesses this information within the `env:Header` element.

The header defines the namespace `am` along with providing the `actor` element, which tells the node that it should route the information somewhere else after it's done processing it. Then within the header there is information for the node that allows the intermediary to act on the data. In this case, a customer Id and a request Id are present to help the initial node do some processing of the data before passing it on. It may, for example, check the person's customer Id to make sure they are still an active customer. The `env:mustUnderstand` attribute means that the node must act on the data in some matter and return an exception if the data is not processed.

```
<?xml version='1.0' ?>
```

```

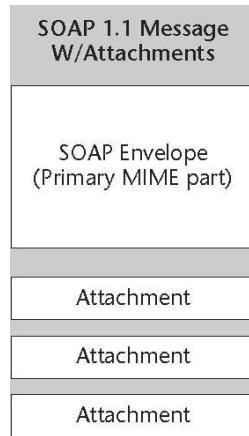
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/SOAP-envelope">
    <env:Header>
        <am:customer
            xmlns:route="http://advocatemedia.com/authenticate"
            env:actor="http://www.w3.org/2001/12/SOAP-envelope/
                actor/next"
            env:mustUnderstand="true">
            <am:custId>4557799</am:custId>
            <am:requestId>12asd-34ccd-23cuden</am:requestId>
        </am:customer>
    </env:Header>

    <env:Body>
        <stockquote:symbolist
            xmlns:stockquote="http://advocatemedia.com/stocks">
            <stockquote:symbol>C</stockquote:symbol>
            <stockquote:symbol>GE</stockquote:symbol>
            <stockquote:symbol>DJI</stockquote:symbol>
        <stockquote:symbolist>
    </env:Body>
</env:Envelope>

```

At this point, the *SOAP* document does not contain information needed for it to bind with a transport protocol.

Typical to the previous example message, the structural format of a *SOAP* message (as per *SOAP* version 1.1 with attachments) contains the following elements: envelope, header (optional), body, attachments (optional). Figure 3.26 represents the structure of a *SOAP* message with attachments. Typically, a *SOAP* message is represented by a *SOAP* envelope with zero or more attachments. The *SOAP* message envelope contains the header and body of the message, and the *SOAP* message attachments enable the message to contain data, which include XML and non-XML data (like text/binary files). In fact, a *SOAP* message package is constructed using the *MIME* Multipart/Related structure approaches to separate and identify the different parts of the message.



*Fig. 3.26 Structure of a SOAP message with attachments.*

The **SOAP envelope** is the primary container of a *SOAP* message's structure and is the mandatory element of a *SOAP* message. It is represented as the root element of the message as *Envelope*. As we discussed earlier, it is usually declared as an element using the XML namespace `http://schemas.xmlsoap.org/soap/envelope/`. As per *SOAP* 1.1 specifications, *SOAP* messages that do not follow this namespace declaration are not processed and are considered to be invalid. Encoding styles also can be defined using a namespace under *Envelope* to represent the data types used in the message. The following code shows the *SOAP envelope* element in a *SOAP* message.

```

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    SOAP-ENV: encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
    <!--SOAP Header elements - -/>
    <!--SOAP Body element - -/>

```

```
</SOAP-ENV:Envelope>
```

The **SOAP header** is represented as the first immediate child element of a SOAP envelope, and it has to be namespace qualified. In addition, it also may contain zero or more optional child elements, which are referred to as SOAP header entries. The SOAP encodingStyle attribute will be used to define the encoding of the data types used in header element entries. The SOAP actor attribute and SOAP mustUnderstand attribute can be used to indicate the target SOAP application node (Sender/Receiver/Intermediary) and to process the Header entries. The following code shows the sample representation of a SOAP header element in a SOAP message.

```
<SOAP-ENV:Header>
  <wiley:Transaction
    xmlns:wiley="http://jws.wiley.com/2002/booktx"
      SOAP-ENV:mustUnderstand="1">
        <keyValue> 5 </keyValue>
    </wiley:Transaction>
</SOAP-ENV:Header>
```

The SOAP header represents a transaction semantics entry using the SOAP mustUnderstand attribute. The mustUnderstand attribute is set to “1”, which ensures that the receiver (URI) of this message must process it. We will look into the mustUnderstand attributes in the next section. SOAP headers also provide mechanisms to extend a SOAP message for adding features and defining high-level functionalities such as security, transactions, priority, and auditing.

A SOAP envelope contains a **SOAP body** as its child element, and it may contain one or more optional SOAP body block entries. The Body represents the mandatory processing information or the payload intended for the receiver of the message. The SOAP 1.1 specification mandates that there must be one or more optional SOAP Body entries in a message. A Body block of a SOAP message can contain any of the following:

- RPC method and its parameters
- Target application (receiver) specific data
- SOAP fault for reporting errors and status information

The following code illustrates a SOAP body representing an RPC call for getting the book price information from [www.wiley.com](http://www.wiley.com) for the book name *Developing Java Web Services*.

```
<SOAP-ENV:Body>
  <m:GetBookPrice xmlns:m="http://www.wiley.com/jws.book.priceList/">
    <bookname xsi:type='xsd:string'> Developing Java Web services</bookname>
  </m:GetBookPrice>
</SOAP-ENV:Body>
```

Like other elements, the Body element also must have a qualified name-space and be associated with an encodingStyle attribute to provide the encoding conventions for the payload. In general, the SOAP Body can contain information defining an RPC call, business documents in XML, and any XML data required to be part of the message during communication.

A SOAP message contains the primary SOAP envelope in an XML format and SOAP **attachments** in any data format that can be ASCII or binary (such as XML or non-text). SOAP attachments are not part of the SOAP envelope but are related to the message.

As the SOAP message is constructed using a MIME multipart/related structure, the SOAP attachment part of the message is contained to a MIME boundary (defined in the Context-Type header). Each MIME part in the structure of the SOAP message is referenced using either Content-ID or Content-Location as labels for the part. Both the SOAP header and body of the SOAP message also can refer to these labels in the message. Each attachment of the message is identified with a Content-ID (typically an hrefattribute using a URL scheme) or Content-Location (a URI reference associated to the attachment).

The following code uses “WileyCoverPage.gif” as an attachment and illustrates the use of the Content-ID (CID) reference in the body of the SOAP 1.1 message using absolute URI-referencing entities labeled for using Content-Location headers.

```

MIME-Version: 1.0 Content-Type: Multipart/Related;
boundary=MIME_boundary; type=text/xml;
start=<http://jws.wiley.com/coverpagedetails.xml> Content-
Description: SOAP message description.

--MIME_boundary-
Content-Type: text/xml; charset=UTF-8
Content-Transfer-Encoding: 8bit
Content-ID: <http://jws.wiley.com/coverpagedetails.xml>
Content-Location: http://jws.wiley.com/coverpagedetails.xml

<?xml version='1.0' ?>
<SOAP-ENV:Envelope
xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
<SOAP-ENV:Body>
<!-- SOAP BODY - -->
<theCoverPage href="http://jws.wiley.com/DevelopingWebServices.gif"/>
<!-- SOAP BODY - -->
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

--MIME_boundary-
Content-Type: image/gif
Content-Transfer-Encoding: binary
Content-ID: <http://jws.wiley.com/DevelopingWebServices.gif>
Content-Location: http://jws.wiley.com/DevelopingWebServices.gif

<!--...binary GIF image... -->
--MIME_boundary-

```

**SOAP Fault.** In a SOAP message, the SOAP Fault element is used to handle errors and to find out status information. This element provides the error and/or status information. It can be used within a Body element or as a Body entry. It provides the following elements to define the error and status of the SOAP message in a readable description, showing the source of the information and its details:

- Faultcode. The faultcode element defines the algorithmic mechanism for the SOAP application to identify the fault. It contains standard values for identifying the error or status of the SOAP application. The namespace identifiers for these faultcode values are defined in <http://schemas.xmlsoap.org/soap/envelope/>. The following fault-code element values are defined in the SOAP 1.1 specification:
  - o VersionMismatch This value indicates that an invalid namespace is defined in the SOAP envelope or an unsupported version of a SOAP message.
  - o MustUnderstand This value is returned if the SOAP receiver node cannot handle and recognize the SOAP header block when the MustUnderstand attribute is set to 1. The MustUnderstand values can be set to 0 for false and 1 for true.
  - o Client This faultcode is indicated when a problem originates from the receiving client. The possible problems could vary from an incorrect SOAP message, a missing element, or incorrect name-space definition.
  - o Server This faultcode indicates that a problem has been encountered during processing on the server side of the application, and that the application could not process further because the issue is specific to the content of the SOAP message.
  - o Faultstring. The faultstring element provides a readable description of the SOAP fault exhibited by the SOAP application.
- Faultactor. The faultactor element provides the information about the ultimate SOAP actor (Sender/Receiver/Intermediary) in the message who is responsible for the SOAP fault at the particular destination of a message.
- Detail. The detail element provides the application-specific error or status information related to the defined Bodyblock.

The following code shows how a SOAP Fault is represented in a SOAP message.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV  
    ="http://schemas.xmlsoap.org/soap/envelope/"  
    SOAP-  
    ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding  
    /">  
  
<SOAP-ENV:Body> <SOAP-ENV:Fault> <faultcode>SOAP-  
    ENV:MustUnderstand</faultcode> <faultstring>Header element  
    missing</faultstring>  
<faultactor>http://jws.wiley.com/GetBookPrice</faultactor>  
<detail>  
<wiley:error xmlns:wiley="http://jws.wiley.com/GetBookPrice">  
<problem>The Book name parameter missing.</problem>  
</wiley:error>  
</detail>  
</SOAP-ENV:Fault>  
</SOAP_ENV:Body>  
</SOAP-ENV:Envelope>
```

The following shows how a SOAP Fault is caused due to server failure.

```
<SOAP-ENV:Fault>  
    <faultcode> SOAP-ENV:Server</faultcode>  
    <faultstring> Server OS Internal failure - Reboot server</faultstring>  
    <faultactor>http://abzdnet.net/net/keysoap.asp</faultactor>  
</SOAP-ENV:Fault>
```

The following shows how a SOAP Fault is caused due to client failure.

```
<SOAP-ENV:Fault>  
    <faultcode>Client</faultcode>  
    <faultstring>Invalid Request</faultstring>  
    <faultactor>http://jws.wiley.com/GetCatalog</faultactor>  
</SOAP-ENV:Fault>
```

The SOAP **mustUnderstand** attribute indicates that the processing of a SOAP header block is mandatory or optional at the target SOAP node. The following example is a SOAP request using mustUnderstand and the response message from the server.

The following shows the request message where the SOAP message defines the header block with a mustUnderstand attribute of 1.

```
<SOAP-ENV:Header>  
    <wiley:Catalog  
    xmlns:wiley="http://jws.wiley.com/2002/  
    bookList"  
    SOAP-ENV:mustUnderstand="1">  
    </wiley:Catalog>  
</SOAP-ENV: Header>
```

The following code is an example response message from the server when the server could not understand the header block where the mustUnderstand is set to 1: is the server-generated fault message detailing the issues with the header blocks using misunderstood and qname (faulting SOAP nodes) and providing a complete SOAP fault.

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV  
    ="http://www.w3.org/2001/06/soap-envelope/"  
    SOAP-ENV:encodingStyle=  
    "http://www.w3.org/2001/06/soap-encoding/"  
    xmlns:fx="http://www.w3.org/2001/06/soap-faults/">  
<SOAP-ENV:Header>  
<fx:misUnderstood qname="wiley:Catalog"  
    xmlns:wiley="http://jws.wiley.com/2002/bookList/" />
```

```

</SOAP-ENV:Header>
<SOAP-ENV:Body> <SOAP-ENV:Fault>
<faultcode>SOAP-ENV:mustUnderstand</faultcode>
<faultstring>Could not understand Header element</faultstring>
</SOAP-ENV:Fault>
</SOAP_ENV:Body>
</SOAP-ENV:Envelope>

```

### 3.3.4.5 SOAP Data Types and Structures

Along with the header, body, and envelope, the *SOAP* standard allows for the representation of certain data types and structures.

**Primitive Types.** The *SOAP* standard does not create new data types for variables, but, rather, uses the data types defined in the XML Schema standard. This allows *SOAP* to represent data in a standard way. Table 3.10 shows the different standard types that a *SOAP* document may represent.

**Table 3.10: The Different Primitive Types Available from XML Schema in the SOAP Standard**

Schema Primitive Type	Description	Example
xsd:int	signed integer value	-9 or 9
xsd:boolean	boolean whose value is either 1 or 0	1 or 0
xsd:string	string of characters	Rocky Mountains
xsd:float or xsd:double	signed floating point number (+,-)	-9.1 or 9.1
xsd:timeInstant	date/time	1969-05-07-08:15
SOAP-ENC:base64	base64-encoded information used for passing binary data SW89IjhhibdOI111QWgdGE within <i>SOAP</i> documents	

**Structs.** A struct is a data structure that you can think of like a container. It is a way of storing several, perhaps vastly different, values in one neat package. In fact, you saw a struct in one of the previous stockquote examples. Here is the snippet that represents a struct.

```

<stockquote:symbolist
  xmlns:stockquote="http://advocatemedia.com/stocks">
  <stockquote:symbol>C</stockquote:symbol>
  <stockquote:symbol>GE</stockquote:symbol>
  <stockquote:symbol>DJI</stockquote:symbol>
</stockquote:symbolist>

```

Modifying this snippet to use the schema's primitive types looks like the following example. (Note that, in the header of the XML document the appropriate namespaces need to be included to use these types)

```

<stockquote:symbolist
  xmlns:stockquote="http://advocatemedia.com/stocks">
  <stockquote:symbol
    xsi:type="string">C</stockquote:symbol>
  <stockquote:symbol
    xsi:type="string">GE</stockquote:symbol>
  <stockquote:symbol
    xsi:type="string">DJI</stockquote:symbol>
</stockquote:symbolist>

```

The information in a struct doesn't have to be so similar. For example, the XML may contain information that describes the author, such as the following.

```

<CRM:AuthorInfo xmlns:CRM="http://www.charlesriver.com/authorinfo">
  <CRM:FirstName xsi:type="string">Brian</CRM:FirstName>
  <CRM:LastName
    xsi:type="string">Hochgurtel</CRM:LastName>
  <CRM:PhoneNumber

```

```

    xsi:type="int">3035551212</CRM:PhoneNumber>
    <CRM:BookTitle
        xsi:type="string">Cross Platform Web Services</CRM:BookTitle>
    </CRM:AuthorInfo>

```

Now moving all the information together may allow an application to handle the data more efficiently.

**Arrays.** The *SOAP* standard also supports the use of arrays. An array is similar to a struct but normally only stores data of the same type, like the following example.

```

<SymbolList
    SOAP-ENC:arrayType="xsd:string[3]">
    <symbol>C</symbol>
    <symbol>GE</symbol>
    <symbol>DJI</symbol>
</SymbolList>

```

This is an alternate, and perhaps more convenient, way to represent data for the Stock Quote Example. Instead of having to define the type for each entry, the array definition allows you to group related data together so you don't have to specify the type each time.

However, it is possible to group unlike values in an array as the following example illustrates.

```

<AuthorInfo SOAP-ENC:arrayType="xsd:ur-type[4]">
    <FirstName xsi:type="string">Brian</FirstName>
    <LastName
        xsi:type="string">Hochgurtel</LastName>
    <PhoneNumber
        xsi:type="int">3035551212</PhoneNumber>
    <BookTitle xsi:type="string">
        Cross Platform Web Services
    </BookTitle>
<AuthorInfo>

```

The definition for the array type, *xsd:ur-type[4]*, indicates that there are four elements in the array of various types. This is much different than the struct type introduced earlier in the chapter.

For one final look at arrays, consider the following example that shows the entire Stock Quote Example using an array. Note that to use the schema type string in the document, the namespaces for schemas must be included in the header.

```

<?xml version='1.0' ?>
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/SOAP-envelope"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:SOAP-ENC="http://schemas.xmlSOAP.org/SOAP/encoding/"
    xmlns:stockquote="http://advocatemedia.com/examples">
    <env:Body>
        <SOAP-ENC:Array SOAP-ENC:arrayType="xsd:string[3]">
            <stockquote:symbol>C</stockquote:symbol>
            <stockquote:symbol>GE</stockquote:symbol>
            <stockquote:symbol>DJI</stockquote:symbol>
        </SOAP-ENC:Array>
    </env:Body>
</env:Envelope>

```

Using the array, the format is slightly cleaner and easier to read, but if you look at the original Stock Quote Example that uses the struct, there really isn't much difference.

**SOAP Encoding.** SOAP 1.1 specifications stated that SOAP-based applications can represent their data either as literals or as encoded values. Literals refer to message contents that are encoded according to the W3C XML

Schema. Encoded values refer to the messages encoded based on SOAP encoding styles. The namespace identifiers for these SOAP encoding styles are defined in <http://schemas.xmlsoap.org/soap/encoding/> (SOAP 1.1) and <http://www.w3.org/2001/06/soap-encoding> (SOAP 1.2).

The SOAP encoding defines a set of rules for expressing its data types. It is a generalized set of data types that are represented by the programming languages, databases, and semi-structured data required for an application. SOAP encoding also defines serialization rules for its data model using an encodingStyle attribute under the SOAP-ENV namespace that specifies the serialization rules for a specific element or a group of elements.

SOAP encoding supports both simple- and compound-type values. Examples are **primitive data types** such as string, integer, decimal, and derived simple data types including enumeration and arrays. The following examples are a SOAP representation of primitive data types:

```
<int>98765</int>
<decimal> 98675.43</decimal>
<string> Java Rules </string>
```

The derived simple data types are built from simple data types and are expressed in the W3C XML Schema.

**Enumeration** defines a set of names specific to a base type. The following code is an example of an enumeration data type expressed in a W3C XML Schema.

```
<xsschema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    elementFormDefault="qualified">
<xselement name="ProductType">
    <xssimpleType base="xsd:string">
        <xsenumeration value="Hardware">
        <xsenumeration value="Software">
    </xssimpleType>
</xselement>
</xsschema>
```

The following is an example of an array data type of **an array of binary data** that is represented as text using base64 algorithms and expressed using a W3C XML Schema.

```
<myfigure xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xmlns:enc=" http://schemas.xmlsoap.org/soap/encoding">
    xsi:type="enc:base64">
    SD334G5vDy9898r32323</myfigure>
```

The **polymorphic accessor** enables programming languages to access data types during runtime. SOAP provides a polymorphic accessor instance by defining an xsi: typeattribute that represents the type of the value. The following is an example of a polymorphic accessor named price with a value type of "xsd:float" represented as follows:

```
<price xsi:type="xsd:float">1000.99</price>
```

And, the XML instance of the price data type will be as follows:

```
<price>1000.99</price>
```

**Compound value types** are based on composite structural patterns that represent member values as structure or array types. The following sections list the main types of compound type values.

The following is an XML Schema of the **Structure** data type representing the “Shipping address” with subelements like “Street,” “City,” and “State.”

```
<xselement name="ShippingAddress"
    xmlns:xs="http://www.w3.org/2001/XMLSchema" >
<xsccomplexType>
```

```

<xs:sequence>
    <xs:element ref="Street" type="xsd:string"/>
    <xs:element ref="City" type="xsd:string"/>
    <xs:element ref="State" type="xsd:string"/>
    <xs:element ref="Zip" type="xsd:string"/>
    <xs:element ref="Country" type="xsd:string"/>
</xs:sequence>
</xs:complexType>
</xs:element>

```

And, the XML instance of the ShippingAddress data type is shown bellow.

```

<e:ShippingAddress>
    <Street>1 Network Drive</Street>
    <City>Burlington</City>
    <State>MA</State>
    <Zip>01803</Zip>
    <Country>USA</Country>
</e:ShippingAddress>

```

The structure also can contain both simple and complex data type values that can reference each other (see bellow). The structure uses the “*href*” attribute to reference the value of the matching element.

```

<e:Product>
    <product>Sun Blade 1000</product>
    <type>Hardware</type>
    <address href="#Shipping"/>
    <address href="#Payment"/>
<e:/Product>
<e:Address id="Shipping">
    <Street>1 Network Drive</Street>
    <City>Burlington</City>
    <State>MA</State>
    <Zip>01803</Zip>
    <Country>USA</Country>
</e:Address>
<e:Address id="Payment">
    <Street>5 Sunnyvale Drive</Street>
    <City>Menlopark</City>
    <State>CA</State>
    <Zip>21803</Zip>
    <Country>USA</Country>
</e:Address>

```

The following is an XML Schema of an **Array** data type representing MyPortfolio—a list of portfolio stock symbols.

```

<xs:schema xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:enc="http://schemas.xmlsoap.org/soap/encoding" >
    <xs:import namespace="http://schemas.xmlsoap.org/soap/encoding" >
        <xs:element name="MyPortfolio" type="enc:Array"/>
    </xs:schema>

```

The XML instance of the MyPortfolio data type is shown bellow.

```

<MyPortfolio xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:enc="http://schemas.xmlsoap.org/soap/encoding"
    enc:arrayType="xs:string[5]">
    <symbol>SUNW</symbol>
    <symbol>IBM</symbol>
    <symbol>HP</symbol>
    <symbol>RHAT</symbol>
    <symbol>ORCL</symbol>

```

```
</MyPortfolio>
```

SOAP encoding also enables arrays to have other arrays as member values. This is accomplished by having the id and href attributes to reference the values. The bellow code shows an example of an XML instance that has arrays as member values.

```
<MyProducts xmlns:xs="http://www.w3.org/2001/XMLSchema"
    xmlns:enc="http://schemas.xmlsoap.org/soap/encoding"
    enc:arrayType="xs:string[] [3]">
    <item href="#product-hw"/>
    <item href="#product-sw"/>
    <item href="#product-sv"/>
    <SOAP-ENC:Array id="product-hw"
        SOAP-ENC:arrayType="xsd:string[3]">
        <item>SUN Blade 1000</item>
        <item>SUN Ultra 100</item>
        <item>SUN Enterprise 15000</item>
    </SOAP-ENC:Array>
    <SOAP-ENC:Array id="product-sw"
        SOAP-ENC:arrayType="xsd:string[2]">
        <item>Sun Java VM</item>
        <item>Sun Solaris OS</item>
    </SOAP-ENC:Array>
    <SOAP-ENC:Array id="product-sv" SOAP-ENC:arrayType="xsd:string[2]">
        <item>Sun Java Center services</item>
        <item>Sun Java Web Services</item>
    </SOAP-ENC:Array>
```

**Partially transmitted arrays** are defined using a SOAP-ENC:offset, which enables the offset position to be indicated from the first element (counted as zero-origin), which is used as an offset of all the elements that will be transmitted. The following listing is an array of size [6]; using SOAP-ENC:offset="4" transmits the fifth and sixth elements of a given array of numbers (0,1,2,3,4,5).

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:string[6]"
    SOAP-ENC:offset="[2]">
    <item> No: 2</item>
    <item> No: 3</item>
    <item> No: 4</item>
    <item> No: 5</item>
</SOAP-ENC:Array>
```

**Sparse arrays** are defined using a SOAP-ENC:position, which enables the position of an attribute to be indicated with an array and returns its value instead of listing every entry in the array. The following shows an example of using a sparse array in an array.

```
<SOAP-ENC:Array SOAP-ENC:arrayType="xsd:int[10]">
    <SOAP-ENC:int SOAP-ENC:position="[0]">0</SOAP-ENC:int>
    <SOAP-ENC:int SOAP-ENC:position="[10]">9</SOAP-ENC:int>
</SOAP-ENC:Array>
```

**Serialization and Deserialization.** In SOAP messages, all data and application-specific data types are represented as XML, and it is quite important to note that there is no generic mechanism to serialize application-specific data types to XML. SOAP implementation provides application-specific encoding for application programming languages (such as Java and C++). It also enables developers to define custom application-specific encoding, especially to handle the data representation required and its data types. This is usually implemented as application- or programming language-specific serialization and deserialization mechanisms that represent application-specific data as XML and XML as application-specific data.

Most SOAP implementations provide their own serialization and deserialization mechanisms and a predefined

XML Schema supporting the SOAP encoding rules and mapping application-specific data types. These serializers and deserializers supporting SOAP encoding rules provide the encoding and decoding of data on runtime by mapping XML elements to target application objects and vice versa. It leverages interoperability between disparate applications using SOAP messages.

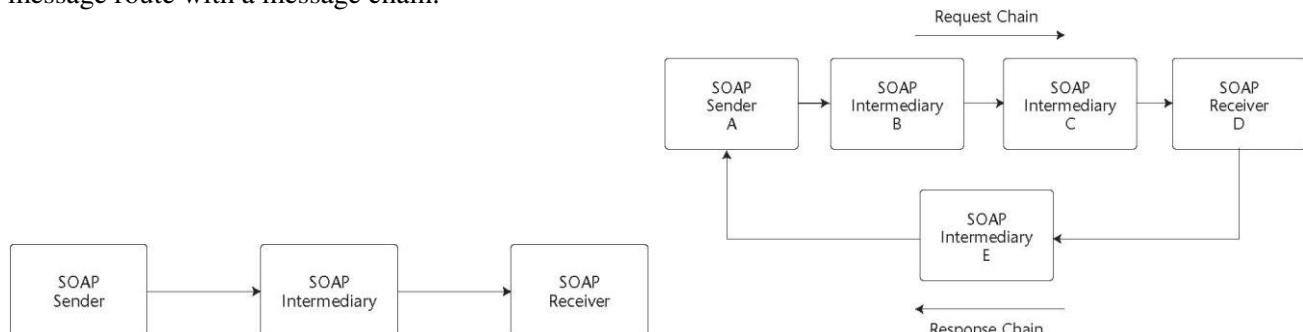
### 3.3.4.6 SOAP Message Exchange Model

Basically, SOAP is a stateless protocol by nature and provides a composable one-way messaging framework for transferring XML between SOAP applications which are referred to as SOAP nodes. These SOAP nodes represent the logical entities of a SOAP message path to perform message routing or processing. In a SOAP message, SOAP nodes are usually represented with an endpoint URI as the next destination in the message. In a SOAP message, a SOAP node can be any of the following:

- SOAP sender. The one who generates and sends the message.
- SOAP receiver. The one who ultimately receives and processes the message with a SOAP response, message, or fault.
- SOAP intermediary. The one who can play the role of a SOAP sender or SOAP receiver.

In a SOAP message exchange model, there can be zero or more SOAP intermediaries between the SOAP sender and receiver to provide a distributed processing mechanism for SOAP messages.

Figure 3.27 (a) represents a basic SOAP message exchange model with different SOAP nodes. In a SOAP message exchange model, the SOAP message passes from the initiator to the final destination by passing through zero to many intermediaries. In a SOAP messaging path, the SOAP intermediaries represent certain functionalities and provide routing to the next message destination. It is important to note that SOAP does not define the actual SOAP senders, intermediaries, and receivers of the SOAP message along its message path or its order of destination. However, SOAP can indicate which part of the message is meant for processing at a SOAP node. Thus, it defines a decentralized message-exchanging model that enables a distributed processing in the message route with a message chain.



**Fig. 3.27 (a) Basic SOAP message exchange model (b) SOAP message exchange model with intermediaries**

Figure 3.27(b) represents an example of a complete message exchange model with a sender, receiver, and its intermediaries. In the previous example, the message originates from Sender A to Receiver D via Intermediaries B and C as a request chain, and then as a response chain the message originates from Receiver D to Sender A via Intermediary E.

SOAP defines **intermediaries** as nodes for providing message processing and protocol routing characteristics between sending and receiving applications. Intermediary nodes reside in between the sending and receiving nodes and process parts of the message defined in the SOAP header. The two types of intermediaries are as follows:

1. Forwarding intermediaries. This type processes the message by describing and constructing the semantics and rules in the SOAP header blocks of the forwarded message.
2. Active intermediaries. This type handles additional processing by modifying the outbound message for the potential recipient SOAP nodes with a set of functionalities.

In general, SOAP intermediaries enable a distributed processing model to exist within the SOAP message exchange model. By using SOAP intermediaries, features can be incorporated like store and forward, intelligent

routing, transactions, security, and logging, as well as other value additions to SOAP applications.

In a SOAP message to represent a target SOAP node, the **SOAP actor** global attribute with a URI value can be used in the Header element. SOAP defines an actor with a URI value, which identifies the name of the SOAP receiver node as an ultimate destination. The following code is an example of a SOAP actor attribute:

```
<SOAP-ENV:Envelope xmlns:SOAP-ENV="http://schemas.xml.org/soap/envelope/"  
    SOAP-ENV:encodingStyle="http://schemas.xml.org/soap/encoding/">  
<SOAP-ENV:Header>  
<b:Name xmlns:t="http://www.wiley.com/BookService/"  
    SOAP-ENV:actor="http://www.wiley.com/jws/" SOAP-ENV  
    :mustUnderstand="1">  
    WebServices</b:Name >  
</SOAP-ENV:Header>  
<SOAP:Body> <m>NewBook xmlns:m="http://www.wiley.com/Books">  
    <BookName>Developing Java Web services</BookName>  
</m>NewBook>  
</SOAP:Body>  
</SOAP:Envelope>
```

Additionally, SOAP defines the actor with a special URI `http://schemas.xmlsoap.org/soap/actor/next`, which indicates a hop-by-hop communication using the header element where the SOAP message is routed via one to many intermediaries before its final destination. The following code is an example of a SOAP message that is forwarded via two SOAP intermediaries before the final receiving node.

```
<SOAP-ENV:Header>  
    <zz:path xmlns:zz="http://schemas.xmlsoap.org/rp/" SOAP-  
        ENV:actor="http://schemas.xmlsoap.org/soap/actor/next" SOAP-  
        ENV:mustUnderstand="1">  
    <zz:action></zz:action>  
    <zz:to>http://www.wiley.com/soap/servlet/rpcrouter</zz:to>  
    <zz:fwd> <zz:via>http://javabooks.congo.com/std/multihop/</zz:via>  
        <zz:via>http://linux.wiley.com/javawebservices/</zz:via>  
    </zz:fwd>  
    </zz:path>  
</SOAP-ENV:Header>
```

### 3.3.4.7 SOAP Communication

SOAP is designed to communicate between applications independent of the underlying platforms and programming languages. To enable communication between SOAP nodes, SOAP supports the following two types of communication models:

**SOAP RPC.** It defines a remote procedural call-based synchronous communication where the SOAP nodes send and receive messages using request and response methods and exchange parameters and then return the values.

**SOAP Messaging.** It defines a document-driven communication where SOAP nodes send and receive XML-based documents using synchronous and asynchronous messaging.

The **SOAP RPC** representation defines a tightly coupled communication model based on requests and responses. Using RPC conventions, the SOAP message is represented by method names with zero or more parameters and return values. Each SOAP request message represents a call method to a remote object in a SOAP server and each method call will have zero or more parameters. Similarly, the SOAP response message will return the results as return values with zero or more out parameters. In both SOAP RPC requests and responses, the method calls are serialized into XML-based data types defined by the SOAP encoding rules.

The following code is an example of a SOAP RPC request making a method call `GetBookPrice` for obtaining a book price from a SOAP server namespace `http://www.wiley.com/jws.book.priceList` using a "book-name" parameter of "Developing Java Web Services".

```

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3c.org/2001/XMLSchema"
    SOAP-ENV:encodingStyle
        ="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
    <m:GetBookPrice
        xmlns:m="http://www.wiley.com/jws.book.priceList">
        <bookname xsi:type='xsd:string'>
            Developing Java Web services</bookname>
        </m:GetBookPrice>
    </SOAP-ENV:Body>
</SOAP-ENV: Envelope>

```

The SOAP message represents the SOAP RPC response after processing the SOAP request, which returns the result of the Get-BookPrice method from the SOAP server namespace <http://www.wiley.com/jws.book.priceList> using a “Price” parameter with “\$50”as its value.

```

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3c.org/2001/XMLSchema"
    SOAP-ENV:encodingStyle
        ="http://schemas.xmlsoap.org/soap/encoding/">
    /> <SOAP-ENV:Body> <m:GetBookPriceResponse
        xmlns:m=" http://www.wiley.com/jws.book.priceList">
        <Price>50.00</Price> </m:GetBookPriceResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

The communication model used above is similar to a traditional CORBA- or RMI-based communication model, except the serialized data types are represented by XML and derived from SOAP encoding rules.

**SOAP Messaging** represents a loosely coupled communication model based on message notification and the exchange of XML documents. The SOAP message body is represented by XML documents or literals encoded according to a specific W3C XML schema, and it is produced and consumed by sending or receiving SOAP node(s). The SOAP sender node sends a message with an XML document as its body message and the SOAP receiver node processes it.

The following code represents a SOAP message and a SOAP messaging-based communication. The message contains a header block InventoryNotice and the body product, both of which are application-defined and not defined by SOAP. The header contains information required by the receiver node and the body contains the actual message to be delivered.

```

<env:Envelope xmlns:env="http://www.w3.org/2001/12/soap-
envelope">
<env:Header>
<n:InventoryNotice
    xmlns:n="http://jws.wiley.com/Inventory">
    <n:productcode>J6876896896</n:productcode>
</n: InventoryNotice>
</env:Header>
<env:Body>
    <m:product xmlns:m="http://jws.wiley.com/product">
        <m:name>Developing Java Web Services</m:name>
        <m:quantity>25000</m:quantity>
        <m:date>2002-07-01T14:00:00-05:00</m:date>
    </m:product>
</env:Body>
</env:Envelope>

```

**SOAP Message Exchange Patterns.** Based on the underlying transport protocol, to enhance the communication and message path model between the SOAP nodes, SOAP chooses an interaction pattern depending upon the communication model. Although it depends upon SOAP implementation, SOAP messages may support the following messaging exchange patterns to define the message path and transmission of messages between SOAP nodes, including intermediaries. It is important to note that these patterns are introduced as part of SOAP 1.2 specifications. The most common SOAP messaging patterns are as follows:

One-way message. In this pattern, the SOAP client application sends SOAP messages to its SOAP server without any response being returned (see Figure 3.28(a)). It is typically found in email messages.

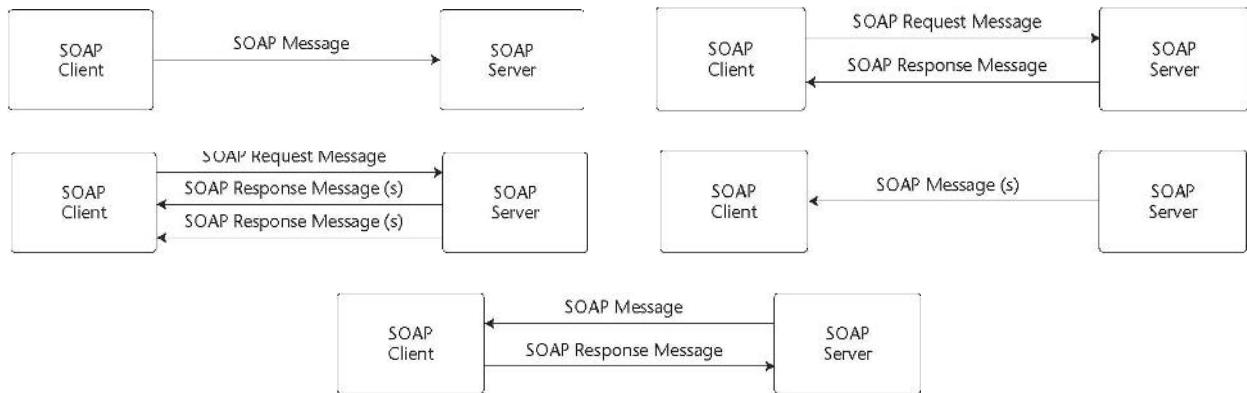
Request/response exchange. In this pattern, the SOAP client sends a request message that results in a response message from the SOAP server to the client (see Figure 3.28 (b)).

Request/N\*Response pattern. It is similar to a request/response pattern, except the SOAP client sends a request that results in zero to many response messages from the SOAP server to the client (Figure 3.28 (c)).

Notification pattern. In this pattern, the SOAP server sends messages to the SOAP client like an event notification, without regard to a response (see Figure 3.28(d)).

Solicit-response pattern. In this pattern, the SOAP server sends a request message to the SOAP client like a status checking or an audit and the client sends out a response message (see Figure 3.28(d)).

Note that the previous patterns can be implemented based on the transport protocols and their supporting communication models.



**Fig. 3.28 (a) One-way message pattern (b) Request/Response pattern (c) Request/N\*Response pattern (d) Notification pattern (e) Solicit-response pattern**

### 3.3.4.8 SOAP Documents and Transport Bindings

In addition to the XML in a *SOAP* request, there is also a header outside of the XML that is specific for the protocol being used, such as HTTP. The information in this header contains the response code, the version of the protocol being used, the content type of the message, and perhaps other vendor-specific information.

The *SOAP* specifications do not specify and mandate any underlying protocol for its communication as it chooses to bind with a variety of transport protocols between the *SOAP* nodes. According to the *SOAP* specifications for binding the framework, the *SOAP* bindings define the requirements for sending and receiving messages using a transport protocol between the *SOAP* nodes. These bindings also define the syntactic and semantic rules for processing the incoming/outgoing *SOAP* messages and a supporting set of message exchanging patterns. This enables *SOAP* to be used in a variety of applications and on OS platforms using a variety of protocols.

Although *SOAP* can potentially be used over a variety of transport protocols, initially the *SOAP* 1.0 specification mandated the use of HTTP as its transport protocol; the later specifications opened their support for other Internet-based protocols like SMTP and FTP. Lately, major *SOAP* vendors have made their implementations available using popular trans-port protocol bindings like POP3, BEEP, JMS, Custom Message-Oriented-Middleware, and proprietary protocols using TCP/IP sockets. *SOAP* uses these protocol bindings as a mechanism for carrying the URI of the *SOAP* nodes. Typically in an HTTP request, the URI indicates the endpoint of the *SOAP* resource

where the invocation is being made.

**SOAP over HTTP.** The use of HTTP as a transport protocol for SOAP communication becomes a natural fit for SOAP/RPC. This enables a decentralized SOAP environment to exist by using the HTTP request/response-based communication over the Internet or an intranet by sending SOAP request parameters in an HTTP request and receiving SOAP response parameters in an HTTP response. Using SOAP over HTTP does not require overriding any existing syntactic and semantic rules of HTTP, but it maps the syntax and semantics of HTTP. By adopting SOAP over HTTP, SOAP messages can be sent through the default HTTP port 80 without requiring and opening other firewall ports. The only constraint while using SOAP over HTTP is the requirement to use the special header tag for defining the MIME type as Content-Type: text/xml.

The following code is an example of an HTTP-based SOAP request for obtaining the book price from <http://jws.wiley.com/GetBookPrice> using bookname as its parameter.

```
POST /GetBookPrice HTTP/1.1
User Agent: Mozilla/4.0 (Linux)
Host: nramesh:8080
Content-Type: text/xml; charset="utf-8"
Content-length: 546
SOAPAction: "/GetBookPrice"
<?xml version="1.0"?>
<SOAP-ENV:Envelope
    SOAP-ENV:encodingStyle=
        "http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
        xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
        <m:getBookPrice
            xmlns:m="http://jws.wiley.com/">
            <bookname xsi:type="xsd:string">
                Developing Java Web Services</bookname>
            </m:getBookPrice>
        </SOAP-ENV:Body>
    </SOAP-ENV:Envelope>
```

The following code is an example of an HTTP-based SOAP response returning the results as the book price from <http://jws.wiley.com/GetBookPrice>.

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 524
Content-Type: text/xml; charset="utf-8"
Date: Fri, 3 May 2002 05:05:04 GMT
Server: Apache/1.3.0
<?xml version="1.0"?>
<SOAP-ENV:Envelope SOAP-ENV:encodingStyle=
    "http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/soap/encoding/"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
    <SOAP-ENV:Body>
        <m:getBookPriceResponse
            xmlns:m="http://jws.wiley.com/GetBookPrice"> <Result
            xsi:type="xsd:string">USD 50.00</Result>
        </m:getBookPriceResponse>
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

In case of errors while processing the SOAP request, the SOAP application will send a response message with HTTP 500 “Internal Server Error” and include a SOAP Fault indicating the SOAP processing error. The SOAP

1.1 specifications define the usage of HTTP as its primary transport protocol for communication. SOAP 1.1 specifications also define the usage of the HTTP extension framework—an extension of the HTTP protocol for adding message extensions, encoding, HTTP-derived protocols, and so on.

How the **HTTP extension framework** is used as a transport binding depends upon the SOAP communication requirements defined by the SOAP nodes. It is similar to HTTP with additional mandatory declarations in the header using an “M-” prefix for all HTTP methods (that is, M-GET, M-POST, and so forth). The following code is a sample header using an HTTP extension framework-based SOAP request.

```
M-POST /GetBookPrice HTTP/1.1
Man: "http://schemas.xmlsoap.org/soap/envelope/";
Content-Type: text/xml; charset="utf-8"
Content-Length: xxxx
SOAPAction: "http://jws.wiley.com/BookPrice#WebServices"

<SOAP-ENV:Envelope>
</SOAP-ENV:Envelope>
```

The following code shows the response header using the HTTP extension framework.

```
HTTP/1.1 200 OK
Ext:
Content-Type: text/xml; charset="utf-8"
Content-Length: xxxx
<SOAP-ENV:Envelope>
</SOAP-ENV:Envelope>
```

In case of errors, the servers force a response message. If the extension declarations do not match the resource, then it responds with a 510 (Not Extended) HTTP status-code. If one or more mandatory extension declarations are present and other following declarations are not true, then it responds with a 505 (HTTP Version Not Supported) HTTP status-code.

**HTTP Request.** The following example shows the Stock Quote *SOAP* Request with its HTTP header. This information tells the server receiving the request where to send the request based on the information here.

The first line states that the information is being posted to the Stock Quotes Web Service using HTTP Version 1.0. The host information on the next line tells the request where on the *World Wide Web* (WWW) the service exists. The Content-Type definition shows what type of information is in the request. In this case, it is XML. Content-length tells how many characters exist in the request. SOAP action contains the namespace for this particular Web Service (<http://www.advocatemedia.com/webservices/>) and the name of the particular method responsible for processing the data (getquote).

```
POST /stockquotes HTTP/1.1
Host: www.advocatemedia.com:80
Content-Type: text/xml; charset=utf-8
Content-Length: 482
SOAPAction: "http://www.advocatemedia.com/webservices/getquote"

<?xml version='1.0' ?>
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/SOAP-envelope"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:SOAP-ENC="http://schemas.xmlSOAP.org/SOAP/encoding/"
    xmlns:stockquote="http://advocatemedia.com/examples">
<env:Body>
    <SOAP-ENC:Array SOAP-ENC:arrayType="xsd:string[3]">
        <stockquote:symbol>C</stockquote:symbol>
        <stockquote:symbol>GE</stockquote:symbol>
        <stockquote:symbol>DJI</stockquote:symbol>
    </SOAP-ENC:Array>
```

```
</env:Body>
</env:Envelope>
```

The header is what makes this example go from an XML document to an actual *SOAP* request.

**HTTP Response.** The following code is a possible response to the Stock Quote Example. The header contains information similar to the request, but not as much information is needed in the response because the connection still knows where the client resides.

The first part of the header indicates that the response comes back via HTTP Version 1.1 and that the status is 200 (which means complete) and OK meaning the data processed correctly. The second line shows that now the request and response are complete the connection is closed. The final two lines are just like the request where Content-Length indicates the number of characters in the message and Content-Type indicates that the response contains XML.

```
HTTP/1.1 200 OK
Connection: close
Content-Length: 659
Content-Type: text/xml; charset=utf-8
<?xml version='1.0' ?>
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/SOAP-envelope"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/SOAP/encoding/"
    xmlns:stockquote="http://advocatemedia.com/examples">
<env:Body>
    <SOAP-ENC:Array SOAP-ENC:arrayType="xsd:int[3]">
        <stockquote:price
            stockquote:symbol="C">53.21</stockquote:price>
        <stockquote:price
            stockquote:symbol="GE">48.00</stockquote:price>
        <stockquote:price
            stockquote:symbol="DJI">9500</stockquote:price>
    </SOAP-ENC:Array>
</env:Body>
</env:Envelope>
```

Notice that the response uses a *SOAP* array to return data.

The example shown in the following listing is a *SOAP* request/response message for **obtaining book price** information from a book catalog service provider. The *SOAP* request accepts a string parameter as the name of the book and returns a float as the price of the book as a *SOAP* response. The *SOAP* message is embedded in an HTTP request for getting the book price information from [www.wiley.com](http://www.wiley.com) for the book *Developing Java Web Services*.

```
POST /BookPrice HTTP/1.1
Host: catalog.acmeco.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 640
SOAPAction: "GetBookPrice"

<SOAP-ENV:Envelope xmlns:SOAP ENV="http://schemas.xmlsoap.org/soap/envelope/">
    xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3c.org/2001/XMLSchema" SOAP-ENV:encodingStyle
        ="http://schemas.xmlsoap.org/soap/encoding/">
    <SOAP-ENV:Header> <person:mail
        xmlns:person="http://acmeco.com/Header/">xyz@acmeco.com </person:mail>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        <m:GetBookPrice
            xmlns:m="http://www.wiley.com/jws.book.priceList">
            <bookname xsi:type='xsd:string'>
```

```

Developing Java Web Services</bookname>
</m:GetBookPrice>
</SOAP-ENV:Body>
</SOAP-ENV: Envelope>

```

The following listing shows the SOAP message embedded in an HTTP response returning the price of the book.

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: 640
<SOAP-ENV:Envelope xmlns:SOAP-
    ENV="http://schemas.xmlsoap.org/soap/envelope/
    xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
    xmlns:xsd="http://www.w3c.org/2001/XMLSchema"
    SOAP-ENV: encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
<SOAP-ENV:Header>
    <wiley:Transaction
        xmlns:wiley="http://jws.wiley.com/2002/booktx" SOAP-
        ENV:mustUnderstand="1"> 5
    </wiley:Transaction>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
    <m:GetBookPriceResponse xmlns:m="
        http://www.wiley.com/jws.book.priceList"> <Price>50.00</Price>
    </m:GetBookPriceResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

The SOAP message contains a SOAP Envelope SOAP-ENV:Envelope as its primary root element, and it relies on defined “XML Namespaces” commonly identified with a keyword xmlns and specific prefixes to identify the elements and its encoding rules. All the elements in the message are associated with SOAP-ENV-defined namespaces. Note that a SOAP application should incorporate and use the relevant SOAP namespaces for defining its elements and attributes of its sending messages; likewise, it must be able to process the receiving messages with those specified namespaces. These namespaces must be in a qualified W3C XML Schema, which facilitates the SOAP message with groupings of elements using prefixes to avoid name collisions. Usually a SOAP message requires defining two basic namespaces: SOAP Envelope and SOAP Encoding. The following list their forms in both versions 1.1 and 1.2 of SOAP.

Additionally, SOAP also can use attributes and values defined in W3C XML Schema instances or XML Schemas and can use the elements based on custom XML conforming to W3C XML Schema specifications. SOAP does not support or use DTD-based element or attribute declarations.

**An Example of a SOAP Error.** If there’s an error, like passing a nonexistent stock symbol, the *SOAP* standard needs to have a mechanism to deal with that. The following example shows what happens with both the XML document and the server header.

The HTTP header indicates that an error occurred during the request. Not only are there the words “Server Error” but also code 500 indicates that there was an error. Within the XML, there are two tags to indicate what happened. The faultcode element contains information that the software can parse and understand what happened. The faultstring element tells the programmer what happened. The following, for example, would be the text that would show up in some sort of pop-up error window in an application.

```

HTTP/1.1 500 Server Error
@dis:Connection: close
Content-Length: 511
Content-Type: text/xml; charset=utf-8
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/SOAP-envelope"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:SOAP-ENC="http://schemas.xmlSOAP.org/SOAP/encoding/">

```

```

<env:Body>
  <env:Fault>
    <env:faultcode>error271</env:faultcode>
    <env:faultstring>No such ticker symbol</env:faultstring>
  </env:Fault>
</env:Body>

```

This example utilizes HTTP as the transport, but the faultcode and faultstring can be matched with any protocol.

The following listing shows a simple but complete **example of a SOAP request for obtaining a stock quote**.

```

<SOAP-ENV:Envelope xmlns:SOAP-ENV="SoapEnvelopeURI"
  SOAP-ENV:encodingStyle="SoapEncodingURI">
  <SOAP-ENV:Header>
  </SOAP-ENV:Header>
  <SOAP-ENV:Body>
    <m:GetLastTradePrice xmlns:m="ServiceURI">
      <tickerSymbol>SUNW</tickerSymbol>
    </m:GetLastTradePrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

This example shows how a SOAP message is encoded using XML and illustrates some SOAP elements and attributes. All SOAP messages must have an `Envelope` element and must define two namespaces: One namespace connotes the SOAP envelope (`xmlns:SOAP-ENV`) and the other indicates the SOAP encoding (`SOAP-ENV:encodingStyle`). SOAP messages without proper namespace specification are considered invalid messages. The `encodingStyle` attribute is important, as it is used to specify serialization rules for the SOAP message. Moreover, there can be no DTD referrals from within SOAP messages.

While optional, the `Header` element when used should be the first immediate child after the `Envelope`. The `Header` element provides a way to extend the SOAP message by specifying additional information such as authentication and transactions. Specifying this additional information as part of the `Header` tells the message recipient how to handle the message.

There are many attributes that can be used in the SOAP `Header` element. For example, the `actor` attribute of the `Header` element enables a SOAP message to be passed through intermediate processes enroute to its ultimate destination. When the `actor` attribute is absent, the recipient is the final destination of the SOAP message. Similarly, many other attributes may be used. However, this chapter does not address these details.

The `Body` element, which must be present in all SOAP messages, must follow immediately after the `Header` element, if it is present. Otherwise, the `Body` element must follow immediately after the start of the `Envelope` element. The `Body` contains the specification of the actual request (such as method calls). The `Fault` element in the SOAP `Body` enables error handling for message requests.

The example shown in the following listing is a SOAP request/response message for obtaining book price information from a book catalog service provider. The SOAP request accepts a string parameter as the name of the book and returns a float as the price of the book as a SOAP response. The SOAP message is embedded in an HTTP request for getting the book price information from [www.wiley.com](http://www.wiley.com) for the book *Developing Java Web Services*.

```

POST /BookPrice HTTP/1.1
Host: catalog.acmeco.com
Content-Type: text/xml; charset="utf-8"
Content-Length: 640
SOAPAction: "GetBookPrice"
<SOAP-ENV:Envelope xmlns:SOAP ENV="http://schemas.xmlsoap.org/soap/envelope/"
  xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3c.org/2001/XMLSchema" SOAP-ENV:encodingStyle
  ="http://schemas.xmlsoap.org/soap/encoding/">
  <SOAP-ENV:Header> <person:mail

```

```

xmlns:person="http://acmeco.com/Header/">xyz@acmeco.com </person:mail>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
  <m:GetBookPrice
    xmlns:m="http://www.wiley.com/jws.book.priceList">
    <bookname xsi:type='xsd:string'>
      Developing Java Web Services</bookname>
    </m:GetBookPrice>
  </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

The following listing shows the SOAP message embedded in an HTTP response returning the price of the book.

```

HTTP/1.1 200 OK
Content-Type: text/xml; charset="utf-8"
Content-Length: 640

<SOAP-ENV:Envelope xmlns:SOAP-
  ENV="http://schemas.xmlsoap.org/soap/envelope/
  xmlns:xsi="http://www.w3c.org/2001/XMLSchema-instance"
  xmlns:xsd="http://www.w3c.org/2001/XMLSchema"
  SOAP-ENV: encodingStyle="http://schemas.xmlsoap.org/soap/encoding"/>
<SOAP-ENV:Header>
  <wiley:Transaction
    xmlns:wiley="http://jws.wiley.com/2002/booktx" SOAP-
    ENV:mustUnderstand="1"> 5
  </wiley:Transaction>
</SOAP-ENV:Header>
<SOAP-ENV:Body>
  <m:GetBookPriceResponse xmlns:m="http://www.wiley.com/jws.book.priceList"> <Price>50.00</Price>
  </m:GetBookPriceResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

The SOAP message contains a SOAP Envelope SOAP-ENV:Envelope as its primary root element, and it relies on defined “XML Namespaces” commonly identified with a keyword xmlns and specific prefixes to identify the elements and its encoding rules. All the elements in the message are associated with SOAP-ENV-defined namespaces. Note that a SOAP application should incorporate and use the relevant SOAP namespaces for defining its elements and attributes of its sending messages; likewise, it must be able to process the receiving messages with those specified namespaces. These namespaces must be in a qualified W3C XML Schema, which facilitates the SOAP message with groupings of elements using prefixes to avoid name collisions. Usually a SOAP message requires defining two basic namespaces: SOAP Envelope and SOAP Encoding.

**SOAP over SMTP.** The *SOAP* standard allows the XML in the message to bind with protocols other than HTTP. This is especially useful if you work on a project that deals with countries that do not have good connections to the Internet. Many times users in these countries, such as many on the African continent, will have the connections time out through HTTP, but with e-mail and *Standard Mail Transport Protocol* (SMTP) the message can take its time getting to the receiver because e-mail gets broken up into several pieces as it works its way through the Internet. HTTP, on the other hand, is looking for an immediate response within a fairly immediate timespan.

To work with SMTP, a *SOAP* document needs to replace the HTTP header with information needed for e-mail. Consider our Stock Quote Request again with a SMTP header.

Rather than having information needed for HTTP, such as Post and the URL of the service, the SMTP *SOAP* header contains an e-mail address, a subject, and a date. A *SOAP* request may also contain unique message Ids. This is just like sending an e-mail to a person except that software generated the e-mail to send to the receiver. In addition, instead of a text message, the application sends the *SOAP* document that contains the XML needed for the Stock Quote Web Service.

```

<?xml version='1.0' ?>
<env:Envelope
  xmlns:env="http://www.w3.org/2001/12/SOAP-envelope">

```

```

xmlns:xsd="http://www.w3.org/1999/XMLSchema"
xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
xmlns:SOAP-ENC="http://schemas.xmlsoap.org/SOAP/encoding/"
xmlns:stockquote="http://advocatemedia.com/examples">
<env:Body>
    <SOAP-ENC:Array SOAP-ENC:arrayType="xsd:int[3]">
        <stockquote:price
            stockquote:symbol="C">53.21</stockquote:symbol>
        <stockquote:price
            stockquote:symbol="GE">48.00</stockquote:symbol>
        <stockquote:price
            stockquote:symbol="DJI">9500</stockquote:symbol>
    </SOAP-ENC:Array>
</env:Body>
</env:Envelope>

```

The response to this request, again, contains the header for SMTP with the exact same response in the *SOAP* document as in previous examples. The subject should change to indicate that some sort of process actually occurred.

The request and response sent via e-mail is read and created by applications of the *SOAP* nodes. You wouldn't want the users of your application decoding some XML in their e-mail inbox in exchange for the data they wanted.

The message is simply returned to the sender with the appropriate XML in the body of the e-mail message.

```

<?xml version='1.0' ?>
<env:Envelope
    xmlns:env="http://www.w3.org/2001/12/SOAP-envelope"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
    xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:SOAP-ENC="http://schemas.xmlsoap.org/SOAP/encoding/"
    xmlns:stockquote="http://advocatemedia.com/examples">
<env:Body>
    <SOAP-ENC:Array SOAP-ENC:arrayType="xsd:int[3]">
        <stockquote:price
            stockquote:symbol="C">53.21</stockquote:symbol>
        <stockquote:price
            stockquote:symbol="GE">48.00</stockquote:symbol>
        <stockquote:price
            stockquote:symbol="DJI">9500</stockquote:symbol>
    </SOAP-ENC:Array>
</env:Body>
</env:Envelope>

```

Having a choice between protocols gives a developer a great deal of flexibility of how to transmit data between nodes.

The use of *SOAP* over *SMTP* permits *SOAP* messages to be enabled with asynchronous communication and supports one-way notifications and document-driven messaging requirements. It also helps *SOAP* messaging where request/response messaging is not a good fit and also where *HTTP* semantics do not apply naturally. The *SOAP* 1.1 specifications define the usage of *SMTP* as a protocol binding for *SOAP* applications, especially where the *HTTP*-based request/ response is not possible and where document-driven messaging is applicable. In case *SOAP* over *SMTP* is used to perform request/response scenarios, it is handled using message correlation techniques by providing unique Message-Id and Reply-To headers. This means that the *SOAP* message will send the request with a Message-Id in the header and the response *SOAP* message will contain an In-Reply-To header containing the originator's Message-Id.

The following code shows an example of a *SOAP* request message using *SOAP* over *SMTP* for obtaining the status information of a purchase order.

To : <[webservices@wiley.com](mailto:webservices@wiley.com)>

From: <nramesh@post.harvard.edu>  
 Reply-To: <nramesh@post.harvard.edu>  
 Date: Tue, 03 May 2002 02:21:00 -0200  
 Message-ID: <1E23B5F132D3EF3C44BCB54532167C5@post.harvard.edu>  
 MIME-Version: 1.0  
 Content-Type: text/xml; charset=utf-8  
 Content-Transfer-Encoding: QUOTED-PRINTABLE

```

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
SOAP-ENV:encodingStyle=
  "http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Body>
  <m:getStatusInfo
  xmlns:m="http://jws.wiley.com/">
    <PurchaseOrderNo>JWS739794-
    04</PurchaseOrderNo>
  </m:getStatusInfo>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
  
```

The response message returning the results will be as shown below. Most SOAP implementations providing the SOAP messaging-based communication model use SMTP to transport SOAP documents between the SOAP nodes.

To: <nramesh@post.harvard.edu>  
 From: <webservices@wiley.com>  
 Date: Tue, 03 May 2002 02:31:00 -0210  
 In-Reply-To: <1E23B5F132D3EF3C44BCB54532167C5@post.harvard.edu>  
 Message-ID: <1E23B5F132D3EF3C44BCB54532167C5@wiley.com>  
 MIME-Version: 1.0  
 Content-Type: TEXT/XML; charset=utf-8  
 Content-Transfer-Encoding: QUOTED-PRINTABLE

```

<?xml version="1.0" encoding="UTF-8"?>
<SOAP-ENV:Envelope
SOAP-ENV:encodingStyle=
  "http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENC="http://schemas.xmlsoap.org/soap/encoding/" xmlns:SOAP-
ENV="http://schemas.xmlsoap.org/soap/envelope/"
xmlns:xsd="http://www.w3.org/2001/XMLSchema"
xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance">
<SOAP-ENV:Body>
  <m:getStatusResponse xmlns:m="http://jws.wiley.com/">
    <status>Product Shipment scheduled - FedEx ID
    866689689689</status>
  </m:getStatusResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
  
```

Note that SMTP may not provide guaranteed message delivery in cases of limitations of message size of the receiving SOAP nodes. Therefore, the SOAP sender nodes may require using custom-delivery receipts and reading the receipts for the email messages they send. The SOAP application developers can use the Internet email messaging servers as the provider to send SOAP messages as email text or attachments. And, it becomes the application developer's responsibility to parse through the email contents and to process the contents of the messages. Some common problems are the result of partial email messages and missing attachments.

**Other SOAP Bindings.** As already noted, SOAP does not mandate any protocol-specific requirements and it can be used with any transport protocols. Using it has a distinct advantage for enabling application integration, inter-application communication, and interoperability. Lately, SOAP application vendors have released their implementations providing support for the SOAP bindings, especially for the most popular industry standard

protocols such as HTTP/S, JMS, and BEEP.

**SOAP over HTTP/SSL.** In addition to using SOAP over HTTP, the SOAP messages can take advantage of using Secure Socket Layer (SSL) for security and other HTTP-based protocol features. SSL enables encrypted data to be securely transmitted between the HTTP client and the server with the use of encryption algorithms. Using SSL with SOAP messages enables the encryption of messages with greater security and confidentiality between the SOAP nodes. It also is possible to add MAC (Media access control) addresses of network card interfaces in the transmitted messages. Using HTTP/SSL requires certificates on both the sending and receiving SOAP nodes. As SOAP does not define security or reliability mechanisms as part of its messages, most SOAP implementations use HTTP/SSL as its transport protocol for secure communication.

**SOAP over JMS.** To enable SOAP messages to communicate with J2EE-based components and messaging applications, most SOAP vendors provide SOAP messaging over JMS (Java Messaging Service) with JMS-compliant MOM providers such as Sun One MQ, Sonic MQ, Websphere MQSeries, and so on. This allows SOAP-based asynchronous messaging and enables the SOAP messages to achieve reliability and guaranteed message delivery using a JMS provider. In this case, the JMS destination queues are represented in the SOAP messages as target destinations. The SOAP nodes use the JMS queue for sending and receiving SOAP requests and SOAP responses. The JMS provider then would implement methods to handle the SOAP message as a payload.

**SOAP over BEEP.** Blocks Extensible Exchange Protocol (BEEP) defines a generic application transport protocol framework for connection-oriented, asynchronous messaging that enables peer-to-peer, client-server, or server-to-server messaging. SOAP over BEEP enables the use of BEEP as a protocol framework that enables SOAP developers to focus on the aspects of the SOAP applications instead of finding a way to establish communication. This means that BEEP takes care of the communication protocol. BEEP, as a protocol, governs the connections, authentication, and sending and receiving of messages at the level of TCP/IP. At the time of this book's writing, the SOAP over BEEP specification is available as an IETF (Internet Engineering Task Force) working draft that can be obtained from <http://beepcore.org/beep/core/beep-soap.jsp>.

### 3.3.4.9 SOAP Security

Security in SOAP messages plays a vital role in access control, encryption, and data integrity during communication. In general, SOAP messages do not carry or define any specific security mechanisms. However, using the SOAP headers provides a way to define and add features enabling the implementation of application-specific security in a form of XML-based metadata. The metadata information can be application-specific information incorporating message security with associated security algorithms like encryption and digital signatures. More importantly, SOAP supports various transport protocols for communication, thus it also is possible to incorporate transport protocol-supported security mechanisms like SSL/TLS for SOAP messages.

The first release of SOAP specifications (SOAP 1.0) did not specify any security-related mechanisms; the following versions of W3C SOAP 1.1 draft specifications were considering enabling security by providing support for implementation of the XML-based security features. At the time of this book's writing, the W3C SOAP Security Extensions specifications were available as a Note to define encryption, authorization, and digital signatures in SOAP messages. But all of the security-related elements are identified using a single namespace identifier using the prefix SOAP-SEC and with an associated URI using <http://schemas.xmlsoap.org/soap/security/>. It also defines the three security element tags <SOAP-SEC: Encryption>, <SOAP-SEC:Signature>, and <SOAP-SEC:Authorization>. Use of these security tags enables the incorporation of encryption, digital signatures, and authorization in SOAP messages.

**SOAP Encryption.** The use of XML-based encryption in SOAP permits secure communication and access control to be implemented by encrypting any element in the SOAP envelope. The W3C XML Encryption WG (XENC) defines the mechanisms of XML encryption in the SOAP messages. In SOAP communication, encryption can be done at the SOAP sender node or at any of the intermediaries in the message path. The following is a sample representation of a SOAP message using XML encryption for encrypting its data elements.

```

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"> <SOAP-ENV:Header>
    <SOAP-SEC:Encryption
        xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/" SOAP-
        ENV:actor="some-URI" SOAP-ENV:mustUnderstand="1"> <SOAP-SEC:EncryptedData>
        <SOAP-SEC:EncryptedDataReference
            URI="#encrypted element"/> </SOAP-SEC:EncryptedData>
        <xenc:EncryptedKey xmlns:xenc=
            "http://www.w3.org/2001/04/xmlenc#" Id="myKey"
            CarriedKeyName="Symmetric Key" Recipient="Bill Allen">
            <xenc:EncryptionMethod
                Algorithm="http://www.w3.org/2001/04/xmlenc#rsa-1_5"/> <ds:KeyInfo
                xmlns:ds= "http://www.w3.org/2000/09/xmldsig#">
                    <ds:KeyName>Bill Allen's RSA Key</ds:KeyName> </ds:KeyInfo>
            <xenc:CipherData>
                <xenc:CipherValue>ENCRYPTED KEY</xenc:CipherValue> </xenc:CipherData>
            <xenc:ReferenceList>
                <xenc:DataReference URI="#encrypted-element"/> </xenc:ReferenceList>
            </xenc:EncryptedKey>
        </SOAP-SEC:Encryption>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        ...
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

This illustrates a SOAP message with a <SOAP-SEC: Encryption> header entry to encrypt data referred to in the SOAP header. It uses a symmetric key for encrypting the body element referred to in the <xenc:EncryptedData> element. The <xenc:EncryptedData> element in the header entry provides the reference to the <xenc: EncryptedData> element and the symmetric key is defined in the <xenc:EncryptedKey> element. On the SOAP receiver node, the receiver decrypts each encrypted element by associating a Decryption-InfoURI, which indicates <xenc:DecryptionInfo> for providing information on how to decrypt it. To find out more information on the syntax and processing rules of representing XML-based encryption, refer to [www.w3.org/TR/xmlenc-core/](http://www.w3.org/TR/xmlenc-core/).

**SOAP Digital Signature.** The use of an XML-based digital signature in SOAP messages provides message authentication, integrity, and non-repudiation of data during communication. The SOAP sender node that originates the message applies an XML-based digital signature to the SOAP body and the receiver node validates the signature. The following is a sample representation of a SOAP message using XML digital signatures.

```

<SOAP-ENV:Envelope
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
    <SOAP-ENV:Header>

        <SOAP-SEC:Signature xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/">
            SOAP-ENV:actor="Some-URI" SOAP-ENV:mustUnderstand="1"> <ds:Signature
                Id="TestSignature"
                xmlns:ds="http://www.w3.org/2000/02/xmldsig#"> <ds:SignedInfo>
            <ds:CanonicalizationMethod Algorithm=
                "http://www.w3.org/2000/CR-xml-c14n20001026"> </ds:CanonicalizationMethod>
            <ds:SignatureMethod Algorithm="http://www.w3.org/2000/09/xmldsig#hmac-sha1"/>
            <ds:Reference URI="#Body"> <ds:Transforms>
            <ds:Transform Algorithm
                ="http://www.w3.org/2000/CR-xml-c14n-20001026"/> </ds:Transforms>
            <ds:DigestMethod Algorithm
                ="http://www.w3.org/2000/09/xmldsig#sha1"/>
                <ds:DigestValue>vAKDSiy987rplkj8ds:DigestValue</ds:DigestValue>
            </ds:Reference> </ds:SignedInfo>
            <ds:SignatureValue>JHJH2374e<ds:SignatureValue> </ds:Signature>
        </SOAP-SEC:Signature>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
        ..
    </SOAP-ENV:Body>
</SOAP-ENV:Envelope>

```

This illustrates a SOAP message with a <SOAP-SEC: Signature> entry applying an XML-based digital signature for signing data included in the SOAP envelope. It uses <ds:CanonicalizationMethod>, <ds:SignatureMethod>, and <ds:Reference>elements for defining the algorithm methods and signing information. The <ds:CanonicalizationMethod> refers to the algorithm for canonicalizing the Signed-Info element digested before the signature. The SignatureMethod defines the algorithm for converting the canonicalized SignedInfo as a SignatureValue. To find more information on the syntax and processing rules of representing XML-based digital signatures, refer to [www.w3.org/TR/xmldsig-core/](http://www.w3.org/TR/xmldsig-core/).

**SOAP Authorization.** Using XML-based authorization in SOAP messages enables the authorization of the SOAP messages using certificates from the originating SOAP sender nodes. SOAP authorization applies an XML-based digital certificate from an independent authorization authority to the SOAP message from the sender. The following is a sample representation of a SOAP message using an XML-based authorization.

```
<SOAP-ENV:Envelope
  xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/">
  <SOAP-ENV:Header>
    <SOAP-SEC:Authorization
      xmlns:SOAP-SEC="http://schemas.xmlsoap.org/soap/security/"
      SOAP-ENV:actor=" actor-URI"
      SOAP-ENV:mustUnderstand="1">
      <AttributeCert xmlns=
        "http://schemas.xmlsoap.org/soap/security/AttributeCert">
        An encoded certificate inserted here as
        encrypted using actor's public key.
      </AttributeCert>
    </SOAP-SEC:Authorization>
    </SOAP-ENV:Header>
    <SOAP-ENV:Body>
    </SOAP-ENV:Body>
  </SOAP-ENV:Envelope>
```

This illustrates a SOAP message with a <SOAP-SEC: Authorization> entry in the SOAP header applying an XML-based authorization to authorize the SOAP message. It uses an <AttributeCert> element to define the certificate from an independent authorization authority. And, it can be encrypted using the receiver node or an actor's public key. On the SOAP receiving node, the actor uses its private key to retrieve the certificate.

### 3.3.5 UDDI

Universal Discovery, Description, and Integration (UDDI) was announced in 2000 as the joint work of Microsoft, IBM, and Ariba. Since its inception, the number of companies that are UDDI sponsors, contributors, liaisons, representatives, and so on has increased enormously, though UDDI is not used as frequently as SOAP or WSDL.

#### 3.3.5.1 Service Lookup through UDDI

Service lookup is an important aspect of distributed systems. The purpose of a service lookup is to provide a directory where services can be advertised. UDDI is the Web Services solution to this problem. In general, a trading cycle involves the following steps (see Figure 3.29):

1. A provider offers a service and wishes to advertise it for clients to use. In order to do so, the provider registers its service with the UDDI registry. The publication request includes information about the offered service, such as the WSDL specification.
2. At a later point in time, a service requestor is looking for a specific functionality. It does an inquiry to the UDDI registry, specifying what it is looking for. When there is a match, the UDDI registry responds with the information regarding a suitable service provider.
3. Once the service requestor knows the details of the service provider, it can bind to the provider. From this moment on, the requestor can interact with the provider.

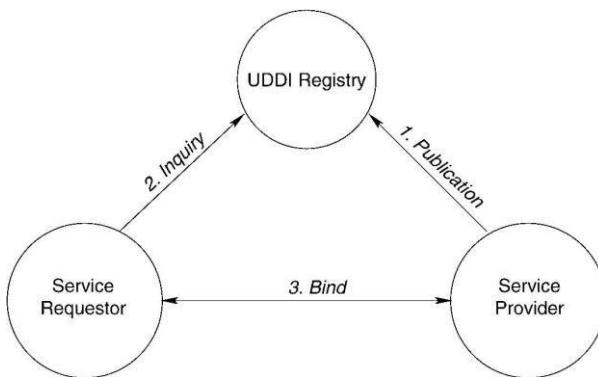


Fig. 3.29 UDDI service trading.

Service mediation has long been a topic of research, and virtually every middleware offers a solution similar to the one outlined above. The first version of UDDI was published in September 2000; it has undergone several revisions since then. The specifications as well as links to other resources are available at <http://www.uddi.org>.

UDDI defines an information model that describes the data maintained at the registry. Conceptually this model contains business information about the entity that provides the service, the type of service being offered, and details on how to invoke the service. All the entries stored in the UDDI registry are classified according to type. For that purpose, UDDI uses several categorization schemes, such as the North American Industry Classification System (NAICS). Categorization organizes the services in a hierarchy and facilitates their discovery.

A UDDI-conformant registry has to understand about two dozen SOAP messages with which clients can interact with the registry. The SOAP interface is used for creating, updating, and querying entries in the registry. Web Services make use of their own standards by using WSDL to describe the interface of a UDDI registry.

UDDI defines the role of a UDDI operator who offers a registry for general public use. A UDDI operator has to offer a conformant interface to its registry. Also a UDDI operator can offer extra services to its client; the UDDI specification mandates that the core registry described by the information model is an exact replica of other operator's registries. Many UDDI operators also offer a Web-based interface to their registries that facilitates service discovery at design time.

#### 3.3.5.2 Registry Standards

UDDI stands for *Universal Discovery, Description, and Integration*. UDDI specification defines a standard way for registering, deregistering, and looking up Web services. UDDI is a standards-based specification for Web service registration, description, and discovery. Similar to a telephone system's yellow pages, a UDDI registry's

sole purpose is to enable providers to register their services and requestors to find services. Once a requestor finds a service, the registry has no more role to play between the requestor and the provider.

UDDI enables dynamic description, discovery, and integration of Web services. A Web service provider registers its services with the UDDI registry. A Web service requestor looks up required services in the UDDI registry and, when it finds a service, the requestor binds directly with the provider to use the service.

The UDDI specification defines an XML schema for SOAP messages and APIs for applications wanting to use the registry. A provider registering a Web service with UDDI must furnish business, service, binding, and technical information about the service. This information is stored in a common format that consists of three parts:

1. White pages— describe general business information such as name, description, phone numbers, and so forth
2. Yellow pages— describe the business in terms of standard taxonomies. This information should follow standard industrial categorizations so that services can be located by industry, category, or geographical location.
3. Green pages— list the service, binding, and service-specific technical information

The UDDI specification includes two categories of APIs for accessing UDDI services from applications:

1. Inquiry APIs— enable lookup and browsing of registry information
2. Publishers APIs— allow applications to register services with the registry

UDDI APIs behave in a synchronous manner. In addition, to ensure that a Web service provider or requestor can use the registry, UDDI uses SOAP as the base protocol. Note that UDDI is a specification for a registry, not a repository. As a registry it functions like a catalog, allowing requestors to find available services. A registry is not a repository because it does not contain the services itself.

### 3.3.5.3 UDDI Overview

UDDI is a repository of Web Services and companies that provides Web Services to the public and other companies. UDDI also contains business information for corporations that work in this arena. The repository concept evolved from the concept of *Business to Business* (B2B) exchanges working furiously to collaborate across the Web. Prior to the UDDI standard, there was not a way for these exchanges to easily find each other, much less the functionality they needed to share. By having a standard place in which a corporation may search for partners, the collaboration process moves forward at a quicker pace.

UDDI provides a directory of web services that is searchable by client. There are two main parts to UDDI: the specification for how to hold all of the information, and the implementation of the specification. In 2001, Microsoft and IBM launched the first two publicly available UDDI registries. The registries allowed everyone interested to search for web services as well as register a new web service to be made searchable, though public UDDI directories never really took off.

UDDI directories are not limited to web services, and can contain services based on a number of protocols and technologies such as telephone, FTP, email, CORBA, SOAP, and Java RMI.

Several major vendors host the UDDI repositories, such as Microsoft and IBM, as a service to the Web Service community. There is currently no charge to use any part of the implementation. Each vendor that hosts a repository ends up replicating with the other partners so that they should contain the same information. Therefore, when you use the repository it shouldn't matter which vendor's implementation you consider.

You contact the repository in one of two ways: either through a Web browser or through *SOAP* requests and responses by using a particular API. This gives you great flexibility because you can either search yourself or create tools to automatically search the repository on a regular basis, looking for functionality you might need.

The UDDI standard confuses many because it encompasses many things, much like the *SOAP* standard. *SOAP* not only describes the contents of a Web Service but also describes how the messages are transmitted and much more.

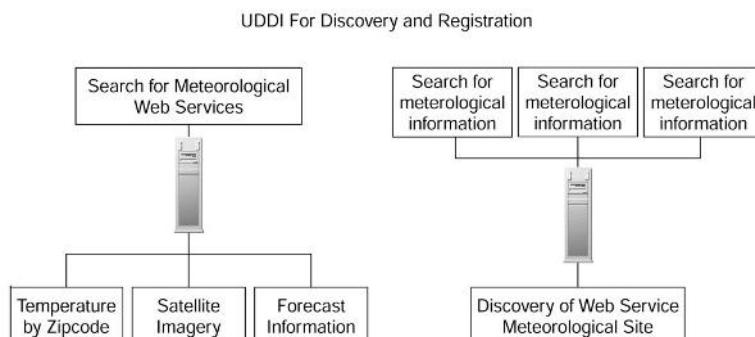
UDDI defines how a repository operates and how the Web Services and corporations are described with XML along with the API for contacting the repository with an application.

The repository of information contains three different sets of data. The first is general contact information, which includes street address and perhaps an individual responsible for fielding inquiries. Searching for the type of service, such as stock quotes, is also available along with information about the type of industry a particular company resides in. By thinking of UDDI as a combination of the white and yellow pages for Web Services and businesses, you begin to get a clearer picture of how to take advantage of UDDI.

Links to WSDL files are often found at a UDDI site, but they do not share any formal relationship. UDDI tells you where the Web Service resides and who sponsors it. On the other hand, WSDL describes the Web Service, including which methods are available. When trying to discover Web Services, you often find that a UDDI site leads you to a WSDL file—that's why the misconception that they are related exists.

If you need to be compatible with Microsoft's .NET environment, look in the repository for Web Services that have a URL for the WSDL file. .NET Web Services need this to create a proxy dll (or a Web reference in Visual Studio .NET) to use the methods in that Web Service.

UDDI Web sites are supported by a group of industry leaders (including HP, IBM, Microsoft, and SAP), who formed a consortium to maintain the UDDI registry system. This system is a database of businesses and the URL of the Web Services they offer, any available WSDL files, and provided business information. Each of the vendors replicates the entries it receives back to the others. Regardless of where you enter the information about your Web Service, it should show up on the other sites as well. Figure 3.30 demonstrates the replication among the different UDDI sites.



**Figure 3.30: The replication of UDDI information amongst the support Web Sites.**

Table 3.11 displays the URLs of the Web sites and the vendors maintaining Web sites for this version.

**Table 3.11: Implementations of UDDI Version 1**

Vendor	URL
Microsoft	<a href="http://uddi.microsoft.com/">http://uddi.microsoft.com/</a>
IBM	<a href="http://www-3.ibm.com/services/uddi/find">http://www-3.ibm.com/services/uddi/find</a>

The main difference between the version 1 and version 2 is an improved *Graphical User Interface* (GUI), along with better security. Table 3.12 shows the URLs and their supporting vendor for Web Sites demonstrating UDDI Version 2. Data in these Beta UDDI Web sites may not be saved when they go operational.

**Table 3.12: Implementations of UDDI Version 2**

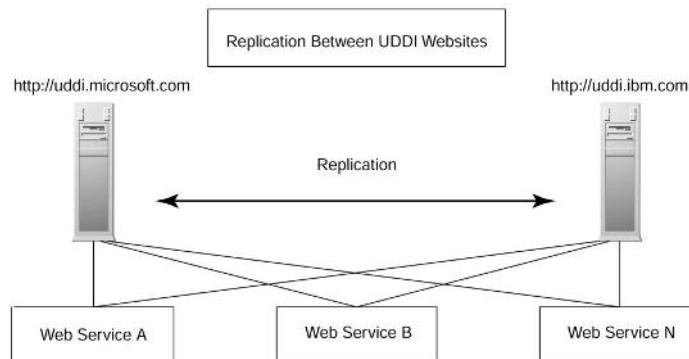
Vendor	URL
Microsoft	<a href="https://uddi.rte.microsoft.com/search/frames.aspx">https://uddi.rte.microsoft.com/search/frames.aspx</a>
IBM	<a href="https://www3.ibm.com/services/uddi/v2beta/protect/registry.html">https://www3.ibm.com/services/uddi/v2beta/protect/registry.html</a>
Hewlett Packard	<a href="https://uddi.hp.com/uddi/index.jsp">https://uddi.hp.com/uddi/index.jsp</a>
SAP	<a href="https://websmp201.sap-ag.de/~form/uddi_discover">https://websmp201.sap-ag.de/~form/uddi_discover</a>

### 3.3.5.3 UDDI Case Studies

A useful implementation of UDDI involves installing an internal UDDI site at a large organization such as a telecom. A telecom could easily have hundreds of internal and external Web Services available to a large audience. By using UDDI, the corporation's IT department takes advantage of a standard that many people are familiar with. In this situation, a new developer or consultant in the organization knows almost immediately where to find information about the services available in the organization. If the repository is kept up to date, it could save an IT department money by reducing training and documentation costs.

Imagine that you're a consultant at a large corporation and you've been charged with bringing data on legacy systems, such as an IBM mainframe, to the public Web site. Instead of hunting down an administrator, finding documentation, or having to create your own documentation, you simply go to the internal UDDI site. You search for the type of Web Service you're looking for, connect your Web page to this Web Service, and then deploy your new Web application.

UDDI's intent is to bring companies who want to do business over the Internet together quicker and in an automatic way. Imagine that you want to set up a Web site to provide your users with meteorological information. Your first step is to identify the information you wish to provide and how you wish to present it. For example, you may wish to provide the temperature, forecast, and barometric pressure based on zip code. For the forecasts, you may want to provide moving images from satellites as a way to enhance your site. Now that you've considered the information you want to provide, you need to determine how you want to display the information. In this case, you decide to provide your meteorological information in the form of Web Services, and then you create Web pages that call the services. By providing Web Services, you gain greater flexibility because Web pages (which may not even be on your Web site), wireless devices, and applications can consume the service. UDDI helps by providing a standard means of promoting your services. Some of the information you provide comes from the home-built weather station in your back yard. The satellite imagery needs to come from another source; by doing a search on a UDDI site you find a Web Service for these images. The UDDI site provides information on where the URL that has the service resides, as well as any contact information needed in case the service requires a fee. Once you complete your site with the Web Services, you can go back to the UDDI registry and create an entry. Currently, using the UDDI repository for promoting services you create is free, but as the repositories grow and cost more money to maintain this could change. Figure 3.31 illustrates how you can use a UDDI repository to search for the services needed and then use it to promote the resulting site.



**Figure 3.31:** Illustrates how you can use a UDDI site to find the Web Services you need and then use the UDDI site to promote the resulting services.

Another aspect to consider is that the Web Service you search for may charge you money each time you use them, so you would want to pass these charges onto your customers. In the case of meteorological services, government agencies often have services available (although not always Web Services) to gather this information for free.

### 3.3.5.4 UDDI and XML

Just as with the *SOAP* standard, UDDI describes multiple aspects—including how UDDI operates and how the XML underneath the information is presented. The information on UDDI sites is stored in a specific XML format. To see one of these XML files, return to <http://uddi.microsoft.com> and do a search on business name for “Sun Microsystems.” The links that come up are for Sun and one of its distributors. You’ll find the following URL: <http://www3.ibm.com/services/uddi/uddiget?businessKey=F293EE60-8285-11D5-A3DA-002035229C64>. This is

the URL that represents Sun's UDDI XML data. Click on that link and you'll see some of the following XML code.

```
<?xmlversion="1.0" encoding="utf-8" ?>
<businessDetail generic="1.0" xmlns="urn:uddi-org:api"
    operator="www.ibm.com/services/uddi"
    truncated="false">
    <businessEntity authorizedName="1000000C5S"
        operator="www.ibm.com/services/uddi"
        businessKey="F293EE60-8285-11D5-A3DA-002035229C64">
        <discoveryURLs>
            <discoveryURL useType="businessEntity">http://www-3.ibm.com/
                services/uddi/uddiget?businessKey=F293EE60-8285-11D5-A3DA-
                002035229C64</discoveryURL>
        </discoveryURLs>
        <name>Sun Microsystems</name>
        <description xml:lang="en">Leading provider of industrial-strength hardware,
            software and services that power the Net</description>
    <contacts>
        <contact useType="Corporate Offices">
            <personName>Sun Microsystems</personName>
        <address>
            <addressLine>901 San Antonio Road</addressLine>
            <addressLine>Palo Alto</addressLine>
            <addressLine>CA 94303</addressLine>
            <addressLine>USA</addressLine>
        </address>
        </contact>
    </contacts>
    <businessServices>
        <businessService serviceKey="02D4B1A0-8291-11D5-A3DA-002035229C64"
            businessKey="F293EE60-8285-11D5-A3DA-002035229C64">
            <name>Products & Solutions</name>
            <description xml:lang="en">Sun's Hardware & Software solutions</description>
            <bindingTemplates>
                <bindingTemplate bindingKey="62A3FA00-8291-11D5-A3DA-002035229C64"
                    serviceKey="02D4B1A0-8291-11D5-A3DA-002035229C64">
                    <accessPoint URLType="http">http://www.sun.com/products-n-solutions/
                </accessPoint>
                <tModelInstanceDetails>
                    <tModelInstanceStateInfo tModelKey="UUID:68DE9E80-AD09-469D-8A37-088422BFBC36" />
                </tModelInstanceDetails>
            </bindingTemplate>
        </bindingTemplates>
    </businessService>
    ...and many more businessService entries
    </businessServices>
</businessDetail>
```

Note that the root element is businessDetail. It contains the operator attribute, which indicates where the entry is hosted. The businessEntity element contains information relevant to the corporation itself. The businessKey attribute is a unique identifier that is created by the UDDI Web site when the entry is made. The name element shows that this entry is for Sun Microsystems, the following element, description, provides information on the corporation, and then several elements give contact information. The binding keys, such as 62A3FA00-8291-11D5-A3DA-002035229C64, are unique identifiers that are created by the UDDI Web site when the entry is created. The businessServices element contains several child elements that are specific to a particular business service. Note that it doesn't necessarily describe a Web Service; in this case, the service entry brings you to a URL on Sun's Web site that describes their products and solutions in general. Luckily, by using the GUIs on the UDDI sites, you can generate this XML just by entering some data into a Web form or by using the UDDI SDK provided by Microsoft.

### 3.3.6 REST

Web architecture teaches us that the way to design an application is to identify its important concepts, model them as *resources*, and assign them Uniform Resource Identifiers (URI). Software *agents*, such as Web browsers or Web applications, then request these resources, specifying their preferred representation formats. The Web server or service responds by transferring the *representation* of the resource to the agent in the format that most closely satisfies the request.

The selection of the best representation is referred to as *content negotiation*. The agent then transitions to its next *state* based on the content of the received resource, which typically contains hyperlinks to other resources. The hyperlinks are then used for subsequent requests, which cause the agent to transition to a new state. This architectural style is referred to as **Representational State Transfer** (REST), a term coined in 2000.

#### 3.3.6.1 Key ideas of REST

Representational State Transfer (REST) is a method of transporting media primarily over the World Wide Web, though it is not restricted to this. It is designed for any hypermedia system—the Web is just the largest. The term defines architectural principles on transfer over systems, but it is loosely tied to transferring data over HTTP without the use of an additional messaging layer, such as SOAP.

There are a few other key ideas in Web architecture. One of the most important is the notion of *hyperlinking*. The representation of a resource will often contain links to other resources. An agent will typically follow these links to retrieve related information. In the context of Web services, this means that the messages exchanged will often contain references to other Web services. A full description of a Web service must also describe the interfaces of these references to other Web services. For example, it is not enough to know that the League Planet schedule Web service returns URIs to teams; it is also necessary to know that these URIs are in fact the endpoints of League Planet team Web services.

Another key idea in REST is the notion of *uniform interface*, which means that there is a standard set of *verbs* or *methods* that can be used to access any resource. In HTTP, the most common methods are PUT, GET, POST, and DELETE, which roughly correspond to the Create, Retrieve, Update, and Delete (CRUD) operations on databases. In practice, most Web applications just use GET and POST.

The proper use of GET has important performance benefits. GET should be used for operations that are *safe*, which means that they are *idempotent* and don't incur any obligations. *Idempotence* means that the result of performing the operation twice is the same as performing it once. For example, in banking, getting your account balance is idempotent, but withdrawing money from it is not. An obligation could be something like having your account charged for the operation. Safe operations admit certain optimizations such as prefetching and caching. For example, a Web browser could prefetch linked pages and cache the results to improve response time and reduce network traffic.

Content negotiation is the mechanism by which an agent can specify the types of resource representation it prefers to receive in response to a request. For example, suppose a Web browser requests an HTML page. Part of the request is an HTTP Accept header that lists the image media types that the Web browser can render, with weightings that indicate its preferences. When the Web application receives the request, it inspects the Accept header and generates a Web page with links to the image media type that best fit the Web browser's preference. A similar mechanism can be used to specify the desired natural language of the response.

Content negotiation applies also to Web services. For example, an AJAX client may prefer a response encoded as JSON as its first choice, then plain XML, and then finally SOAP, since this is the order that minimizes its parsing time. The client includes an Accept header in its requests indicating that it accepts these three media types and then assigns them suitable preferences. In this way, the AJAX client receives the response in the format that is most efficient for it to process if the Web service provides it, but can still function if the Web service provides other acceptable formats. The nice thing about this architecture is that the Web service can be upgraded at a later date to improve performance and the clients will automatically benefit without any modification on their end. For example, suppose League Planet provides REST style Web services initially using plain XML, but then finds after reviewing the server logs that many clients prefer JSON. The service can then be upgraded to also provide JSON, and the existing clients will experience a performance boost without changing a line of their code.

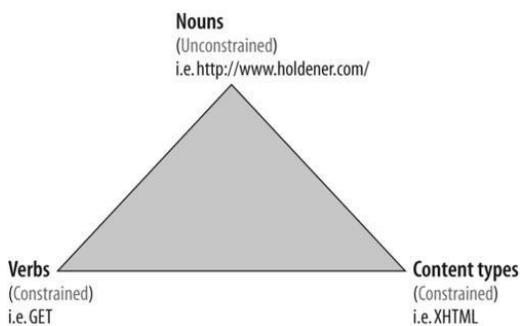
### 3.3.6.2 Key components

The key components for a *RESTful* design are as follows:

- The state and functionality of an application are separated into different resources.
- Every resource shares a consistent method for the transfer of state between resources.
- Every resource is addressable using hypermedia syntax.
- It is a protocol that is stateless, cacheable, client/server-based, and layered.

The World Wide Web is a perfect example of a RESTful implementation, as it can be made to conform to the REST principles. HTML has implicit support for hyperlinks built into the language. HTTP has a consistent method (GET, POST, PUT, and DELETE) to access resources from URIs, methods, status codes, and headers. HTTP is stateless (unless cookies are utilized), has the ability to control caching, utilizes the notion of a client and a server, and is layered so that no layer can know anything about another except for its immediate conversation (connection).

For REST applications, the resource that defines its interface is constrained so that fewer types are defined on the network and more resources are defined. You can think of the interface as verbs, and the content types and resource identifiers as nouns. REST defines the nouns to be unconstrained so that clients do not need knowledge of the whole resource. Figure 3.32 shows the REST triangle of nouns, verbs, and content types.



*Fig. 3.32. The REST triangle of nouns, verbs, and content types*

The architecture we use for the applications we build ultimately depends on the web services with which the architecture will interact. When we know that the only interface to a web service is through SOAP, it might naturally be easier to define the architecture of the application to follow SOA. On the other hand, if the web service is RESTful in nature, building the application to follow REST would more likely be in order. Even more likely, one component of an application will be built one way and another component a different way. The application on the whole takes whatever architectural style is needed for the project (client/server, Model-View-Controller – MVC, etc.), while the components follow their own models to be as efficient as possible.

### 3.3.6.3 REST Style Web Services

The REST architectural style is directly applicable to Web services. Some vendors, such as Amazon, offer both SOAP and REST interfaces. Amazon was finding in 2003 that 85 percent of their Web service usage was via the REST interface. This overwhelming preference for REST versus SOAP is undoubtedly due to the fact that the main use of the Amazon Web service is for providing product links on Web pages. Nevertheless, REST style interfaces are easier to use in this type of application and deserve to be given serious consideration when designing Web services in general.

Most SOAP 1.1 engines employ a single URL that acts like a router for service requests. The SOAP engine examines the request to determine the operation and then invokes the service implementation associated with it. Furthermore, SOAP 1.1 over HTTP always uses POST, so all operations are treated as unsafe. WSDL 1.1 is not much better. In addition to the SOAP 1.1 binding, WSDL 1.1 defines two HTTP bindings, one for GET and another for POST. This means you cannot describe a service that has a combination of safe and unsafe operations. Nor can you always use the correct HTTP method for any given operation since PUT, DELETE, and so forth are not supported. Finally, WSDL 1.1 provides no way to describe messages that refer to other Web services, that is, no support for hyperlinking.

So we see that SOAP 1.1 and WSDL 1.1 are somewhat REST-hostile. Nevertheless, these specifications do

provide the basis for a large and rich set of additional specifications collectively referred to as WS-\*<sup>1</sup>. These include WS-Security, WS-Reliability, and WS-Addressing to name a few. The way to think about WS-\* is that it defines a way to flow Web services messages over multiple transport hops involving a combination of protocols. For example, an enterprise Web service might receive a request over HTTP and then place it on a message queue. This is the domain of SOA.

Although SOA is undoubtedly useful in many contexts, it is overkill in others. For example, suppose you want to build an AJAX client. You need to get XML data from somewhere. Why not use a Web service? In this situation, you'd like the XML to be very easy to process, so SOAP encoding is ruled out. Document/literal style is much more appropriate. But maybe XML is even too complex here. Perhaps JSON is a better representation. Still, this is programmatic access, and even though you are not using SOAP or even XML, you'd like a well-defined interface you can program to.

Fortunately, the combination of SOAP 1.2 and WSDL 2.0 brings the world of Web services into much better alignment with REST architectural principles. SOAP 1.2 supports the use of GET for requests. WSDL 2.0 allows the description of safe operations, has a much improved HTTP binding, and includes support for describing messages that refer to other Web services; that is, hyper-linking between Web services can be described. As we enter the so-called Web 2.0 technology era, we could see a unification of WS-\* and REST style Web services based on SOAP 1.2 and WSDL 2.0.

### **3.3.7 Other Industry Standards Supporting Web Services**

**ebXML** defines a global electronic marketplace where enterprises find one another and conduct business process collaborations and transactions. It also defines a set of specifications for enterprises to conduct electronic business over the Internet by establishing a common standard for business process specifications, business information modeling, business process collaborations, collaborative partnership profiles, and agreements and messaging. In the Web services model, ebXML provides a comprehensive frame-work for the electronic marketplace and B2B process communication by defining standards for business processes, partner profile and agreements, registry and repository services, messaging services, and core components. It complements and extends with other Web services standards like SOAP, WSDL, and UDDI. In particular:

- ebXML Business Process Service Specifications (BPSS) enable busi-ness processes to be defined.
- ebXML CPP/CPA enables business partner profiles and agreements to be defined, and it provides business transaction choreography.
- ebXML Messaging Service Handler (MSH) deals with the transport, routing, and packaging of messages, and it also provides reliability and security, a value addition over SOAP.
- ebXML registry defines the registry services, interaction protocols, and message definitions, and ebXML repository acts as storage for shared information. The ebXML registries register with other reg-istries as a federation, which can be discovered through UDDI. This enables UDDI to search for a business listing point to an ebXML Registry/Repository.
- ebXML Core components provide a catalogue of business process components that provide common functionality to the business com-munity. Examples of such components are Procurement, Payment, Inventory, and so on.
- Many industry initiatives and standards supporting Web services are cur-rently available and many more will be available in the future. The most prominent initiatives to embrace Web services standards are described in the following sections.

The Web Services Choreography Interface, or **WSCI**, is an initiative that defines the flow of messages exchanged in a particular process of Web services communication. It describes the collective message flow model among Web services by pro-viding a global view of the processes involved during the interactions that occur between Web services communication. This facilitates the bridging of business processes and Web services by enabling Web services to be part of the business processes of an organization or spanning multiple organi-zations. For more information about WSCI, go to the Sun XML Web site at [www.sun.com/software/xml](http://www.sun.com/software/xml).

The Web Services Flow Language, or **WSFL**, is an XML-based language initiative for describing Web services compositions. These compositions are categorized as flow models and global models. Flow models can be used for modeling business processes or workflows based on Web services, and global models can be used for

modeling links between Web services interfaces that enable the interaction of one Web service with an operation to another Web service interface. Using WSFL compositions support a wide range of interaction patterns between the partners participating in a business process, especially hierarchical interactions and peer-to-peer interaction between partners.

The Directory Services Markup Language, or **DSML**, defines an XML schema for representing directory structural information as an XML document, and it allows the publishing and sharing of directory information via Internet protocols like HTTP, SMTP, and so forth. DSML does not define the attributes for the directory structure or for accessing the information. A DSML document defines the directory entries or a directory schema or both, and it can be used on top of any industry standard directory protocols like LDAP. DSML defines the standard for exchanging information between different directory services and enables interoperability between them. Bowstreet originally proposed DSML as a standard and later it received support from leading vendors like IBM, Oracle, Sun Microsystems, Microsoft, and so on. For more information about DSML standards, visit [www.dsml.org](http://www.dsml.org).

Similar to WSFL, **XLANG** defines an XML-based standard specification for defining business process flows in Web services. It also defines a notation for expressing actions and complex operations in Web services. Microsoft developed the XLANG specification and it has been implemented in Microsoft BizTalk server 2000, especially for handling Enterprise Application Integration (EAI) and B2B communication.

The Business Transaction Protocol specification (**BTP**) provides a support for Web services-based distributed transactions enabling the underlying transaction managers to provide the flexibility of handling XA-compliant, two-phase commit transaction engines. BTP is an OASIS initiative that facilitates large-scale business-to-business (B2B) deployments enabling distributed transactions in Web services.

The XML Encryption, or **XML ENC**, is an XML-based standard for securing data by encryption using XML representations. In Web services, it secures the exchange of data between the communicating partners.

The XML Key Management System, or **XKMS**, is an XML-based standard for integrating public key infrastructure (PKI) and digital certificates used for securing Internet transactions, especially those used in Web services. XKMS consists of two parts: the XML Key Information Service Specification (X-KISS) and the XML Key Registration Service Specification (X-KRSS). The X-KISS specification defines a protocol for a trust service that resolves public key information contained in XML-SIG elements. The X-KRSS describes how public key information is registered.

The XML Encryption, or **XML DSIG**, is an XML-based standard for specifying XML syntax and processing rules for creating and representing digital signatures. In Web services, an XML digital signature helps XML-based transactions by adding authentication, data integrity, and support for non-repudiation to the data during data exchange among the communicating partners.

The Extensible Access Control Markup Language, or **XACML**, is an XML-based standard for specifying policies and rules for accessing information over Web-based resources. In Web services, XACML sets the rules and permissions on resources shared among the communicating partners. XACML is one of the security initiatives made by the OASIS security services technical committee.

The Security Assertions Markup Language, or **SAML**, defines an XML-based framework for exchanging authentication and authorization information. SAML uses a generic protocol consisting of XML-based request and response message formats, and it can be bound to many communication models and transport protocols. One of the key objectives of SAML is to provide and achieve single sign-on for applications participating in Web services. SAML is an initiative from the security services technical committee of OASIS.

### 3.3.8 Web Services with Apache SOAP

The Apache Group provides a free *SOAP* library to allow you to create and deploy Web Services across multiple platforms because it utilizes Java as the underlying language. This is an advantage if you have an environment containing a mix of operating systems because Java works on many platforms.

The main differences between Apache *SOAP* and the Web Services found in the *.NET* environment includes ease of use and installation. Microsoft's installation of either the *.NET Framework SDK* or *Visual Studio.NET* is seamless. You install it and you're ready to begin, and the code is at a very high level. With Apache *SOAP*, the installation is far more complex. It is not difficult, but you must pay close attention to the installation instructions. If you miss one step or you don't have the environment set up right, you will find the installation of Apache *SOAP* to be somewhat frustrating. The other difference is that coding for both creating a Web Service and calling a service happens at a lower level where you are more aware of the XML being used in the *SOAP* requests. Microsoft hides many of these implementation details from you.

There are several libraries and software components to install. The first piece to install is the Apache Group's Java container, *Tomcat*, which allows you to host the *SOAP* server in addition to servlets and JSPs. The next step after *Tomcat* involves installing all the necessary libraries including *SOAP*, *JavaMail*, and others. Once the libraries are installed, the CLASSPATH environment variable needs to be installed correctly.

#### 3.3.8.1 Creating and Deploying an Apache SOAP Web Service

The Apache *SOAP* implementation is powerful because Java gives you a lot of options for utilizing Web Services because you have the ability to compile code on multiple platforms with Java. Microsoft *.NET*, on the other hand, is limited for now to just *Windows*. Thus, the Apache *SOAP* library offers a great deal more flexibility.

**The Web Service.** The following code is the Java version of our SimpleStockQuote example. It starts out with the package declaration. Note that the statement package samples.simplestock indicates that this example exists in a directory structure /samples/simplestock somewhere in your CLASSPATH.

The first two import statements are standard libraries for handling URLs and Java input and output. The second two import statements, which include org.w3c.dom.\* and org.xml.sax.\* bring in functionality for different models of parsing XML. The javax.xml.parsers.\* allows a program to obtain and manipulate an XML parser, and finally org.apache.soap.util.xml.\* allows access to the functionality in the Apache *SOAP* library. You don't really need to worry about which import statements are used because these examples use the imports that all Apache Web Service code needs.

After all the import statements, the code finally gets to the functionality that allows the Web Service to work. First, the class SimpleStockQuote is defined. Note that the file must be named SimpleStockQuote.java to compile it. Then there is the simple code for the method getTestQuote, which returns the same values as all the other Stock Quote examples in this book.

```
package samples.simplestock;
import java.net.URL;
import java.io.*;
import org.w3c.dom.*;
import org.xml.sax.*;
import javax.xml.parsers.*;
import org.apache.soap.util.xml.*;

public class SimpleStockQuote {
    public String getTestQuote (String symbol) throws Exception {
        if (symbol.equals("C") ) {
            return "55.95";
        } else {
            return "-1";
        }
    }
}
```

Now that the service is written, you can compile it by typing javac SimpleStockQuote.java. This will create the SimpleStockQuote.class file which is the compiled byte code Apache *SOAP* needs to execute the Web Service.

**Deploying the Service.** To deploy an Apache *SOAP* Web Service, creating an XML file called a “deployment descriptor” is necessary to register a service with the server. It identifies the class, methods, and the name used to expose the functionality via Web Services.

The following XML example is the deployment descriptor for the Web Service created in the previous example. The root element defines the needed namespace and the ID attribute describes the name needed to call the methods in that service. When you write the client for this example, the URN called will be simple-stock-quote. The provider element defines the exposed method and the type of Web Service. In this case, you used Java. The scope attribute has three possible values: request, session, and application. Table 3.13 explains each option.

**Table 3.13: The Three Options for the Scope Attribute in the Deployment Descriptor**

Option	Purpose
Request	The <i>SOAP</i> implementation removes the object once the request is complete.
Session	The values a particular object holds only last the life of the HTTP session.
Application	The object lasts as long as the servlet that is managing the <i>SOAP</i> object is in service.

The isd:java tag describes the location of the class. Remember that SimpleStockQuote was part of the samples.simplestock package.

Finally, the fault listener tag describes the class that handles any *SOAP* fault returned from the server. In this case, it is the default class provided by the Apache group. If you needed to, you could add your own fault listener to provide custom error handling for an application.

```
<isd:service xmlns:isd="http://xml.apache.org/xml-soap/deployment"
              id="urn:simple-stock-quote">
    <isd:provider type="java"
                  scope="Application"
                  methods="getTestQuote">
        <isd:java class="samples.simplestock.SimpleStockQuote"/>
    </isd:provider>
    <isd:faultListener>
        org.apache.soap.server.DOMFaultListener
    </isd:faultListener>
</isd:service>
```

Now that you have a deployment descriptor, you must use tools provided with the Apache *SOAP* library to actually deploy the Web Service.

**Deploying SimpleStockQuote.** Now that you have a deployment descriptor, you can execute the following class to complete deployment.

```
org.apache.soap.server.ServiceManagerClient.
```

Start *Tomcat* and allow it to load all the different webapps that are installed. To install the SimpleStockQuote example, execute the following from the directory where you have the code.

```
java org.apache.soap.server.ServiceManagerClient
      http://localhost:8080/soap/servlet/rpcrouter deploy
      DeploymentDescriptor.xml
```

When you execute the deploy command, no information comes back to let you know that the deployment was successful other than that there was no error. Looking in the *SOAP* Admin tools will confirm the installation of SimpleStockQuote. By going to <http://localhost:8080/soap/admin/index.html> you can see what functionality has been deployed. Click on the “List” icon and you should see the SimpleStockQuote Web Service as the only

deployed Web Service. Remember that in the deployment descriptor you gave the Web Service the following unique identifier: urn:simple-stock-quote. This is how the Admin tool lists the various services.

Another option for listing the available services on a particular system is to execute the following command from the DOS prompt.

```
java org.apache.soap.server.ServiceManagerClient  
http://localhost:8080/soap/servlet/rpcrouter list
```

The response is a list of deployed services and yours should simply look like this:

```
Deployed Services:  
urn:simple-stock-quote
```

**Securing the deployment of Web Services.** With the default configuration of Apache *SOAP*, anyone can deploy Web Services to your system, and that's very dangerous from a security perspective. This is fine when the server is in a development environment where it is protected by a firewall and possibly a subnet, but when you put your system into production you want to be sure that you control who installs Web Services.

By changing one configuration file and adding another, you can turn the remote installation of Web Services off. The first step is to modify the *SOAP* distribution's web.xml file. This is the configuration file for the servlets that support the RPC router and the "Admin" tool. If you installed *Tomcat* into the suggested directory, the web.xml file resides in the [tomcat-directory]\web-apps\soap\WEB-INF directory. When opened, this file should look like the following.

```
<web-app>  
  <display-name>Apache-SOAP</display-name>  
  <description>no description</description>  
  <servlet>  
    <servlet-name>rpcrouter</servlet-name>  
    <display-name>Apache-SOAP RPC Router</display-name>  
    <description>no description</description>  
    <servlet-class>  
      org.apache.soap.server.http.RPCRouterServlet  
    </servlet-class>  
    <init-param>  
      ...
```

This is just a sample of first part of the file because you only need to modify the beginning. Right after the description tag, add the following information.

```
<context-param>  
  <param-name>ConfigFile</param-name>  
  <param-value>c:/xmlapache/soap.xml</param-value>  
</context-param>
```

This creates a pointer to a configuration file, soap.xml, where you can tell Apache *SOAP* to do different things. In the soap.xml file, add the following XML code.

```
<soapServer>  
  <serviceManager>  
    <option name="SOAPInterfaceEnabled" value="false" />  
  </serviceManager>  
</soapServer>
```

This disables your ability to add Web Services from the command line. If you try to add a service, you will get the following response.

```
Ouch, the call failed:  
Fault Code = SOAP-ENV:Server.BadTargetObjectURI  
Fault String = Unable to determine object id from
```

call: is the method element namespaced?

**Viewing the SOAP Messages.** In Microsoft's implementation of .NET Web Services, the *SOAP* messages are part of the Web page the service generates. With Apache *SOAP*, there is a means of looking at the messages of Apache *SOAP* and other implementations using the *TCPTunnelGui* tool. This tool provides you with a means of viewing the actual message the client and server are exchanging. To execute the tool, use the following command:

```
java org.apache.soap.util.net.TcpTunnelGui 8000 localhost 8080
```

The *TCPTunnelGui* acts as a proxy to the Web Service requests and response. It takes the request, prints it to the screen, and then forwards it to the server. It does the same thing with the response. Thus, the number "8000" in the previous example is the port it is listening to, localhost is the server it forwards requests to, and 8080 is the port that server is listening to. Then you must modify your client to send requests to port 8000 to make this work.

For closer examination, here is the complete *SOAP* request for the SimpleStockQuote example.

```
Content-Type: text/xml;
charset=utf-8 Content-Length: 461
SOAPAction: ""
<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
    ENV="http://schemas.xmlsoap.org/soap/envelope/"
        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">
<SOAP-ENV:Body>
    <ns1:getTestQuote xmlns:ns1="urn:simple-stock-quote"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <symbol xsi:type="xsd:string">C</symbol>
    </ns1:getTestQuote>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

<t>And the response from the Server for SimpleStockQuote:

```
HTTP/1.0 200 OK
Content-Type: text/xml;
charset=utf-8 Content-Length: 483
Set-Cookie2: JSESSIONID=3bymuj1261;Version=1;Discard;Path="/soap"
Set-Cookie: JSESSIONID=3bymuj1261;Path=/soap
Date: Thu, 08 Aug 2002 00:45:49 GMT
Servlet-Engine: Tomcat Web Server/3.2.4 (JSP 1.1; Servlet 2.2; Java
    1.4.0;
Windows 2000 5.0 x86; java.vendor=Sun Microsystems

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:SOAP-
    ENV="http://schemas.xmlsoap.org/soap/envelope/"

        xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
        xmlns:xsd="http://www.w3.org/2001/XMLSchema">

<SOAP-ENV:Body>
    <ns1:getTestQuoteResponse xmlns:ns1="urn:simple-stock-quote"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding/">
        <return xsi:type="xsd:string">55.95</return>
    </ns1:getTestQuoteResponse>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

Now that you understand how to create, deploy, and monitor an Apache *SOAP* Web Service, you should be ready to create consumers in Java.

### 3.3.8.2 Creating Web Service Consumers in Java, Java Servlets, and JSP Pages

A consumer, as mentioned in previous chapters, is a piece of software or a Web page that actually utilizes a Web Service. In this chapter, the consumers will be a command line Java application, a servlet, and a *Java Server Page* (JSP).

**Command Line Application.** The first three import statements are just general Java functionality needed to read and write files along with handling URLs. All the functionality in this example occurs in the main method. The code starts off by defining the *SOAP* encoding style and the URL to call the Web Service. It then puts the one command line argument into the string symbol. Then the actual call to the Web Service is created. The particular Web Service called is defined as urn:simple-stock-quote. Remember that this is the unique identifier created in the deployment descriptor shown earlier in the chapter.

The vector params is the structure into which the value of symbol is placed. Once in the vector, the SetParams method adds params to the call. Next, the response object receives the values returned by the invoke method of the call object. Finally, there is some error handling to catch errors generated by the *SOAP* request. If there is no error, the following snippet displays the result.

```
Parameter result = resp.getReturnValue ();
System.out.println (result.getValue ());
```

The complete code example:

```
import java.io.*;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.rpc.*;

public class GetQuote {
    public static void main (String[] args)
        throws Exception {

        String encodingStyleURI = Constants.NS_URI_SOAP_ENC;
        //specify the URL of the Web Service
        URL url =new URL
        ("http://localhost:8080/soap/servlet/rpcrouter");
        //Get the string passed in at the command line.
        String symbol = args[0];
        System.out.print("****" + symbol + "****");

        //Create the call to get a specific Web Service
        Call call = new Call ();
        call.setTargetObjectURI ("urn:simple-stock-quote");
        call.setMethodName ("getTestQuote");
        call.setEncodingStyleURI(encodingStyleURI);

        //Create the vector that has the string value.
        Vector params = new Vector ();
        //Add the string to the vector
        params.addElement (new Parameter("symbol", String.class, symbol, null));
        call.setParams (params);
        //grab the response
        Response resp = call.invoke ( url,"" );

        //if there's a fault, catch it.
        //else display the result
        if (resp.generatedFault ()) {
            Fault fault = resp.getFault ();
            System.err.println("Generated fault: " + fault);
        } else {
            Parameter result = resp.getReturnValue ();
            System.out.println (result.getValue ());
        }
    }
}
```

```

    }
}
}
}
```

**Java Swing Example.** Creating a GUI that calls the Web Service in Java is just slightly more work than the command line application shown in the previous example. You need to create the different components that make up the GUI, and then use an event listener to listen for actions happening within the application.

The first step is to import all the proper libraries. In this case, you need to import not only the classes for the Apache *SOAP* but also for *Swing* and *AWT*. *Swing* and *AWT* are the libraries within Java that allow you to create *Windows* applications that are cross platform. The *AWT* library is more operating-system-dependent than *Swing*, but it still supports many of the *Swing* objects.

The next step is to define the class called *WebSvcGui*. Notice the keywords extends and implements in the class definition. Extends indicates that the class inherits either all or some of the *Frame* class whereas implements indicates that the class adheres to the interface defined in *ActionListener*.

After the class definition, the constructor for the class creates the GUI. Notice the definitions for all the frames and buttons used in the GUI. There are two *JTextField* definitions and this is where values are read from and written to. The action listener calls the Web Service's static method *callService()*. The action listener waits for something to happen to one of the objects in the GUI, which is the *valueButton* object. The *callService()* method is static so you don't need to instantiate another copy of the object to call the method from the constructor.

Notice that the definition for the *callService* method is doing more error handling than the command line application. The *Swing* classes require this extra error handling so the GUI knows how to react. Other than the handling this, the code to call the Web Service isn't different than the previous example.

Then, main method creates an instance of *WebSvcGui* and makes it visible. Remember that all actions that occur in the GUI are taken care of by an action listnener.

```

import java.awt.*;
import java.awt.event.*;
import javax.swing.*;
import java.io.*;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class WebSvcGui extends Frame implements
ActionListener {

public WebSvcGui () {
    //name the window
    super("Web Service Gui Example");
    setSize(400,116);

    //create panel and size
    JPanel myPanel = new JPanel();
    myPanel.setLayout(new GridLayout(3,2,5,5));
    JLabel symbol = new JLabel("Symbol:", JLabel.RIGHT);
    final JTextField symbolField = new JTextField(10);
    JLabel result = new JLabel("Result", JLabel.RIGHT);
    final JTextField resultField = new JTextField(10);

    //add a button
    JButton valueButton = new JButton("Get Value");

    //create a listener for the button.It's listening for the button to be clicked
    //Call the Web Service from the listener
    valueButton.addActionListener(new ActionListener() {
        public void actionPerformed(ActionEvent ev) {

```

```

        String symbol = symbolField.getText();
        //call the static Web Service method
        String returned = WebSvcGui.callService(symbol);
        resultField.setText(returned);
    }
});

myPanel.add(symbol);
myPanel.add(symbolField);
myPanel.add(result);
myPanel.add(resultField);
myPanel.add(valueButton);
//add the panel to the gui.
add(myPanel, BorderLayout.SOUTH);
}

public void actionPerformed(ActionEvent ae) {
    System.out.println(ae.getActionCommand());
}
//static so we can call it without creating an WebSvcGui
//object
static String callService(String symbol) {

    String encodingStyleURI = Constants.NS_URI_SOAP_ENC;
    URL url = null;
    //set up URL to call
    try {
        url = new URL
            ("http://homer:8080/soap/servlet/rpcrouter");
    } catch (MalformedURLException e){
        System.out.println("Error:" + e);
    }

    //set up to the call to the Web Service
    Call call = new Call ();
    call.setTargetObjectURI ("urn:simple-stock-quote");
    call.setMethodName ("getTestQuote");
    call.setEncodingStyleURI(encodingStyleURI);
    Vector params = new Vector ();
    params.addElement (new Parameter("symbol", String.class, symbol, null));
    call.setParams (params);
    Response resp;

    //try to call the Web Service, if successful return the
    //value
    try {
        resp = call.invoke ( url, "" );
        if (resp.generatedFault ()) {
            Fault fault = resp.getFault ();
            System.out.println("fault: " + fault );
            return("-1");
        } else {
            Parameter result = resp.getReturnValue ();
            return(result.getValue().toString());
        }
    } catch (SOAPException e) {
        System.out.println("Error:" + e); }
}

public static void main(String args[]) {
    WebSvcGui myGui = new WebSvcGui();
    myGui.setVisible(true);
}
}

```

**Servlet Consumer.** A servlet is a chunk of Java code that executes on the server side for a request from a Web browser in a container such as *Tomcat*. A servlet implementation routes requests to the various Web Services within Apache *SOAP*.

The main difference between the following example and the previous is that a servlet has no main method. It executes as an extension of *HttpServlet* which the extends statement defines. In addition, there are more include files because you need all the functionality that any other servlet would normally have.

After the definition of the class *SimpleStockClient*, two methods must be defined: *doGet* and *doPost*. These are necessary for the servlet to understand requests from a browser.

A browser can send two different types of requests to a Web site. The Post method is one way a browser sends information, usually through a form in the body of the submission. The other is a Get and this occurs when a URL contains data in the query string.

In this example, the *doGet* method contains all the functionality, and then the *doPost* method just routes the request to *doGet*, as the following code snippet shows.

```
public void doPost(HttpServletRequest request, HttpServletResponse response)
    throws IOException, ServletException
{
    doGet(request, response);
}
```

Modifying this example to do different things dependent on the type of request simply involves modifying the *doPost* method.

This example must handle HTML because the response from this example is sent to the browser, and you can see there are several *println* statements sending HTML back to the browser. This makes the servlet code hard to read and this is one of the reasons JSP was created. There are some JSP examples later in the chapter.

```
import java.io.*;
import java.net.*;
import java.text.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import javax.servlet.*;
import javax.servlet.http.*;

//define the servlet
public class SimpleStockClient extends HttpServlet {

    //handle a get request
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        //start the HTML response
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.println("<head>");

        String title = "Get Simple Stock Quote";
        out.println("<title>" + title + "</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<body>");
        out.println("<h1>" + title + "</h1>");

        String encodingStyleURI = Constants.NS_URI_SOAP_ENC;
```

```

//the java code should now resemble the small app shown previously
URL url =new URL("http://homer:8080/soap/servlet/rpcrouter");
String symbol = "C";
out.println("****" + symbol + "****");

Call call = new Call ();
call.setTargetObjectURI ("urn:simple-stock-quote");
call.setMethodName ("getTestQuote");
call.setEncodingStyleURI(encodingStyleURI);
Vector params = new Vector ();
params.addElement (new Parameter("symbol",
String.class, symbol, null));
call.setParams (params);
Response resp;
//catch any errors.
try {
    resp = call.invoke ( url, "" );
    if (resp.generatedFault ()) {
        Fault fault = resp.getFault ();
        out.println("<h2><font color=\"red\">
Generated fault: " + fault + " </font></h2>");
    } else {
        Parameter result = resp.getReturnValue ();
        out.print("Result of Web Service call: ");
        out.println (result.getValue ());
    }
} catch ( SOAPException e ) {
    out.println(" unable to call Web Service " + e);
}
out.println("</body>");
out.println("</html>");
}
}

```

The next example is a little more functional because it allows you to enter a value and submit it to the server to get a response. This is just a matter of adding some more println statements that have the appropriate HTML code for a form. Then an if statement tests to see if symbol has a value. If it does, it calls the appropriate Web Service with the value for symbol.

```

import java.io.*;
import java.net.*;
import java.text.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class SimpleStockClientForm extends HttpServlet {
    public void doGet(HttpServletRequest request, HttpServletResponse response)
        throws IOException, ServletException
    {
        response.setContentType("text/html");
        PrintWriter out = response.getWriter();

        out.println("<html>");
        out.println("<head>");

        String title = "Get Simple Stock Quote";
        out.println("<title>" + title + "</title>");
        out.println("</head>");
        out.println("<body bgcolor=\"white\">");
        out.println("<body>");
        out.println("<h1>" + title + "</h1>");

```

```

out.println("<P>");

//This begins the definition of the form so
//that you can send data to the servlet.
out.println("<form action=\"./SimpleStockClientForm\" method=POST>");

out.print("Please Enter a Stock Symbol:");
out.println("<input type=text size=20 name=symbol>");
out.println("<input type=submit>");
out.println("</form>");

//most of the code is similar to the last servlet example.
String encodingStyleURI = Constants.NS_URI_SOAP_ENC;

URL url =new URL("http://homer:8080/soap/servlet/rpcrouter");
String symbol = request.getParameter("symbol");

//if a value has been sent by the form go ahead
//process the data.
if(symbol != null && symbol.length() != 0) {
    Call call = new Call ();
    call.setTargetObjectURI ("urn:simple-stock-quote");
    call.setMethodName ("getTestQuote");
    call.setEncodingStyleURI(encodingStyleURI);
    Vector params = new Vector ();
    params.addElement (new Parameter("symbol",
        String.class, symbol, null));
    call.setParams (params);
    Response resp;
    try {
        resp = call.invoke /* router URL */ url, /* actionURI */ "";
        if (resp.generatedFault ()) {
            Fault fault = resp.getFault ();
            out.println("<h2><font color=\"red\">
Generated fault: " + fault + "</font></h2>");
        } else {
            Parameter result = resp.getReturnValue ();
            out.print("Result of Web Service call: ");
            out.println (result.getValue ());
        }
    } catch ( SOAPException e ) {
        out.println(" unable to call Web Service " + e);
    }
}
out.println("</body>");
out.println("</html>");
}

public void doPost(HttpServletRequest request, HttpServletResponse response)
throws IOException, ServletException
{
    doGet(request, response);
}
}
}

```

Servlets are very useful for delivering the output of complex Java code to a browser, but if you have a lot of HTML to deliver the coding becomes difficult. JSP pages allow you to separate coding from content, and thus may provide you a little more flexibility as a Web Services consumer.

**Using JSP to Code a Comsumer.** JSP simplifies server-side Java coding because it allows you to develop separate complex Java code from the actual display of information. Servlets display HTML with a series of println statements, but JSP pages mix the HTML with the Java code.

It is interesting to note that the first time JSP code executes the Java server, such as Apache Tomcat, actually generates servlet code from the template the JSP page defined. The servlet code then compiles and executes, and it sends the response back to the browser.

JSP can use Java code, but it also implements many HTML-like tags to perform many of the same functions as the Java code. Take a look at the first JSP consumer and compare it to one of the servlet examples, and you'll find that they are very different.

**Simple JSP Consumer.** The following code example shows how different JSP code is. The import statements are replaced with `<%@ page import .. %>`. Instead of using `println` statements to display HTML, the HTML mixes in with the JSP elements. Then the Java code that calls the Web Service is the same code used in the previous examples. The difference here is that the code appears between the `<% %>` tags. Code that appears between these tags are referred to as *scriptlets*.

```
<%@ page import = "java.io.*, java.net.*,
    java.util.*, org.apache.soap.*, org.apache.soap.rpc.*, java.util.*" %>
<HTML>
    <HEAD>
        <TITLE>JSP Web Services Page</TITLE>
    </HEAD>
    <BODY>
        <H2>JSP Web Services Page</H2>
        <%
        String encodingStyleURI = Constants.NS_URI_SOAP_ENC;
        URL url =new URL ("http://homer:8080/soap/servlet/rpcrouter");
        String symbol = "C";

        Call call = new Call ();
        call.setTargetObjectURI ("urn:simple-stock-quote");
        call.setMethodName ("getTestQuote");
        call.setEncodingStyleURI(encodingStyleURI);
        Vector params = new Vector ();
        params.addElement (new Parameter("symbol",
            String.class, symbol, null));
        call.setParams (params);
        Response resp;
        try {
            resp = call.invoke ( url, "" );
            if (resp.generatedFault ()) {
                Fault fault = resp.getFault ();
                out.println("<h2><font color=\"red\">Generated
                fault: " + fault + " </font></h2>");
            } else {
                Parameter result = resp.getReturnValue ();
                out.print("Result of Web Service call: ");
                out.println (result.getValue ());
            }
        } catch ( SOAPException e ) {
            out.println(" unable to call Web Service " + e);
        }
        %> </BODY>
    </HTML>
```

**JSP with Include Directive.** The goal of JSP is to separate content from functionality. In the last example, much of the Java code still mixes in with the JSP and HTML elements. One way around having all the Java within the scriptlet tags `<% and %>` is to use a `jsp:include` directive and have all the Java code appear in the separate file.

Consider the following simple JSP page.

```
<HTML>
    <HEAD>
        <TITLE>JSP Include Example</TITLE>
    </HEAD>
    <H2>JSP Include Example</H2>
    <jsp:include page="IncludeExample.jsp" flush="true"/>
</HTML>
```

It simply includes a JSP page, which contains the following code.

```
<%@ page import = "java.io.*, java.net.*, java.util.*,
org.apache.soap.*, org.apache.soap.rpc.*, java.util.*" %>
<%
String encodingStyleURI = Constants.NS_URI_SOAP_ENC;
URL url =new URL ("http://homer:8080/soap/servlet/rpcrouter");
String symbol = "C";

Call call = new Call ();
call.setTargetObjectURI ("urn:simple-stock-quote");
call.setMethodName ("getTestQuote");
call.setEncodingStyleURI(encodingStyleURI);
Vector params = new Vector ();
params.addElement (new Parameter("symbol", String.class, symbol, null));
call.setParams (params);
Response resp;
try {
    resp = call.invoke (url, "");
    if (resp.generatedFault ()) {
        Fault fault = resp.getFault ();
        out.println("<h2><font color=\"red\">Generated
fault: " + fault + " </font></h2>");
    } else {
        Parameter result = resp.getReturnValue ();
        out.print("Result of Web Service call: ");
        out.println (result.getValue ());
    }
} catch ( SOAPException e ) {
    out.println(" unable to call Web Service " + e);
}
}
```

This is the same code that appears in the first JSP example, but it appears in a separate JSP file.

**Beans and JSP.** Another way of separating complex Java code from the internal workings of a JSP is to put that code in a Bean. A Bean is a library of Java code that is similar to a dll in Microsoft implementations. It is accessible to JSP pages via *Tomcat*, and by using set and get methods the Bean and JSP page are able to communicate.

Tomcat looks for Beans in the classes subdirectory of WEB-INF. Within that directory you need to create a subdirectory and place your Bean within it. The name of that directory (or an entire path from the class's directory) needs to be defined with a package directive at the beginning of the Bean. If it isn't in a subdirectory, Tomcat attempts to load the class file as a servlet.

By using Beans, the JSP pages you implement become very clean because all the complex Java code is hidden. In the following JSP example, there is not scriptlet code. All the communications with Java is done with the jsp:usebean, jsp:setProperty, and the jsp:getProperty tags.

The jsp:usebean tag has a opening and closing element so that the properties of the Bean can be set. You'll notice that the Bean only has one property set, but there could be several.

Now that the values are set, it is just a matter of using jsp:getProperty and the browser displays the value.

```
<HTML>
<HEAD>
    <TITLE>Bean Example</TITLE>
</HEAD>
<BODY>
<H2>XPlatform Bean Example</H2>
<jsp:useBean id="getQuote"
            class="XPlatform.getServiceBean">
<jsp:setProperty name="getQuote"
                 property="symbol"
```

```

        value="C" />
</jsp:useBean>
Value:<jsp:getProperty name="getQuote"
                           property="price"/>
</BODY>
</HTML>
```

The Bean code still has much of the same Web Service code in it that has been used throughout the examples found in this chapter, but now it appears within a get method named getPrice. By using standard get and set methods the JSP easily communicates with the Bean. Notice that there aren't any getSymbol or setPrice methods because they are not needed for this example. This is the complete code example for the Bean.

```

//bean must appear in WebInf/classes/XPlatform
package XPlatform;
import java.net.*;
import java.util.*;
import org.apache.soap.*;
import org.apache.soap.rpc.*;

public class getServiceBean {

    public String symbol = null;
    public String price = null;

    public void setSymbol(String s) {
        symbol = s;
    }
    public String getPrice() {
        String encodingStyleURI = Constants.NS_URI_SOAP_ENC;
        URL url = null;
        //set up URL to call
        try {
            url = new URL ("http://homer:8080/soap/servlet/rpcrouter");
        } catch (MalformedURLException e){
            System.out.println("Error:" + e);
        }
        //set up to the call to the Web Service
        Call call = new Call ();
        call.setTargetObjectURI ("urn:simple-stock-quote");
        call.setMethodName ("getTestQuote");
        call.setEncodingStyleURI(encodingStyleURI);
        Vector params = new Vector ();
        params.addElement (new Parameter("symbol",
        String.class, symbol, null));
        call.setParams (params);
        Response resp;
        String returnValue = null;
        //try to call the Web Service, if successful return the value
        try {
            resp = call.invoke ( url, "" );
            if (resp.generatedFault ()) {
                Fault fault = resp.getFault ();
                System.out.println("fault: " + fault );
                price = "-1";
            } else {
                Parameter result = resp.getReturnValue ();
                price = result.getValue().toString();
            }
        } catch (SOAPException e) {
            System.out.println("Error:" + e);
        }
        return(price);
    }
}
```

### 3.3.9 Web Services with Apache Axis

Apache Axis is an implementation of the SOAP ("Simple Object Access Protocol") submission to W3C. Axis is a reliable and stable base to implement Java Web services, and there are many companies who use Axis for web services support in their products. Moreover, there is a very active user community too for Axis. Axis comes in two forms, Axis 1.x and Axis 2. Axis 2 architecture is recent as compared to the predecessor, and is a redesign of Axis 1.x supporting SOAP 1.2, REST and more. There are many production deployments in 1.x code base too.

Apache *Axis* is an attempt by the Apache Group to create an open source Web Services tool that is compatible with other available systems such as Microsoft's .NET. Several features that were missing from Apache *SOAP* appear in this release. This release is a complete rewrite of Java Web Services. Apache *Axis* has its roots in IBM's Web Services toolkit. At one point, the project transferred from IBM to the Apache Group.

Unlike software you purchase, open source software usually requires several installation steps before you are able to utilize the software. Apache *Axis* is no different, but you'll find the setup slightly easier than Apache *SOAP*. Apache *Axis* utilizes *Tomcat*.

#### 3.3.9.1 Creating and Deploying an Axis WS: Differences between Apache Axis and SOAP

The coding of a Web Service with *Axis* doesn't look that much different than a *SOAP* Web Service. The main differences are the import statements and how you compile the service for deployment. The following examples show an *Axis* Web Service and its XML deployment descriptor.

**The Web Service.** Consider the following code for *Axis* that contains the code for the SimpleStockExample class used in previous chapters.

```
public class SimpleStockExample {  
    public float getTestQuote(String symbol)  
        throws Exception {  
        if ( symbol.equals("C") ) {  
            return( (float) 55.95 );  
        } else {  
            return( (float) -1 );  
        }  
    }  
}
```

This is just a simple program with no import or package statements. You can compile this in the directory where you created it to ensure that it is syntactically correct, but when you're ready to deploy, change the extension of the Java (SimpleStockExample.java) file from .java to .jws (SimpleStockExample.jws) and copy it into [tomcat-directory]\webapps\axis\. With *Tomcat* running, point your browser to the following URL: <http://localhost:8080/axis/SimpleStockExample>. By pointing the browser to that URL, you compile the code into a Web Service.

Apache Axis creates its own WSDL for each Web Service, and just like Microsoft's .NET, this WSDL is available by simply putting "WSDL" in the query string of the URL like this:

<http://localhost:8080/axis/SimpleStockExample?WSDL>

If, for some reason, this method of deployment doesn't work for you, you can create a directory under the *Axis* directory in the *Tomcat* directory structure and create a Web Service that looks like the following.

```
package samples.SimpleStock;  
import org.w3c.dom.Document;  
import org.w3c.dom.Element;  
import org.w3c.dom.NodeList;  
  
import javax.xml.parsers.DocumentBuilder;  
import javax.xml.parsers.DocumentBuilderFactory;  
  
public class SimpleStockExample {
```

```

public float getTestQuote(String symbol)
    throws Exception {
    if ( symbol.equals("C") ) {
        return( (float) 55.95 );
    } else {
        return( (float) -1 );
    }
}

```

This last example is more like what was found with the Apache *SOAP* library, and also requires a deployment descriptor. Using this method may be more work, but it may give you more flexibility because a deployment descriptor allows you to specify certain meta information that you don't get when using a .jws file.

The **deployment descriptor** in Apache *Axis* uses different syntax than its predecessor in *SOAP*, but the idea remains the same. It's an XML document that describes how the *Axis* server should deploy the service if it is necessary to specify additional information beyond just creating a .jws file.

In the following example, note that the deployment element starts off by giving this deployment description a name and describes the namespaces used in the document. The service element attributes name the service and describe the type or provider that makes the service available. The child elements, which are both parameter, describe the class available to the service and the methods to be accessed. In this case "\*" gets used to indicate that all the methods are available.

```

<deployment
    name="load"
    xmlns="http://xml.apache.org/axis/wsdd/"
    xmlns:java="http://xml.apache.org/axis/wsdd/providers/java">
    <service name="SimpleStockExample"
        provider="java:RPC">
        <parameter name="className"
            value="samples.SimpleStock.SimpleStockExample"/>
        <parameter name="allowedMethods" value="*"/>
    </service>
</deployment>

```

Save the XML in a file called deploy.wsdd and use the following command to set up the Web Service with *Axis*:

```
java org.apache.axis.client.AdminClient deploy.wsdd
```

Now, if you go look at the deployed Web Services in *Axis*' admin GUI, you should see that SimpleStockExample is now considered a deployed Web Service.

In Apache *SOAP* there was no easy means of providing WSDL, but with *Axis* the WSDL for the example is found by adding "WSDL" in the query string like this:

```
http://localhost:8080/axis/services/SimpleStockExample?wsdl
```

**Undeploying the SimpleStockExample.** An XML file is also required to remove or undeploy a Web Service from the *Axis* server.

The following XML example undeploys the SimpleStockExample. The root element is undeployment, whose attributes define the name of this undeployment descriptor and the namespace for the XML file. The service child element describes which class should no longer be served by *Axis*.

```

<undeployment name="load" xmlns="http://xml.apache.org/axis/wsdd/">
    <service name="SimpleStockExample"/>
</undeployment>

```

Then, by executing the following command, the AdminClient reads the undeploy.wsdd to determine which class or classes to remove.

```
java org.apache.axis.client.AdminClient undeploy.wsdd
```

**Relaxing deployment rules.** Unlike Apache *SOAP*, *Axis* does not come with remote administration installed. It only allows you to administer from the local machine. If you wish to allow remote administration to occur so an entire development team has access to deployment, the server-confi.wsdd file needs to be modified. This file contains directives for the entire *Axis* server, and by adding the following to the AdminService directive, the deployment of Web Services from remote machines is enabled.

```
<service name="AdminService" provider="java:MSG">
    ...Other directives
    <parameter name="enableRemoteAdmin" value="true"/>
</service>
```

The entire entry might look like the following.

```
<service name="AdminService" provider="java:MSG">
    <parameter name="allowedMethods" value="AdminService"/>
    <parameter name="enableRemoteAdmin" value="false"/>
    <parameter name="className" value="org.apache.axis.utils.Admin"/>
    <parameter name="sendXsiTypes" value="true"/>
    <parameter name="sendMultiRefs" value="true"/>
    <parameter name="sendXMLDeclaration" value="true"/>
    <namespace>http://xml.apache.org/axis/wsdd/</namespace>
    <parameter name="enableRemoteAdmin" value="true"/>
</service>
```

Having remote administration rights turned on by default was a major security problem with the *SOAP* library because many people wouldn't know it was turned on. This way users cannot deploy Web Services to your server without your knowledge.

**Using TCP Monitor.** *TCP Monitor* is an updated version of the *TCP Tunnel* GUI. It is a tool that allows you to view the requests and responses to a particular server. It acts as a proxy so that it intercepts requests and responses, prints them to the screen, and then forwards them to the appropriate server or client. To use this tool, execute the following command.

```
java org.apache.axis.utils.tcpmon 8000 localhost 8080
```

Click on the tab that says "Port 8080" and you'll see any Web Service request and responses to that port.

Unlike the *TCP Tunnel* GUI, *TCP Monitor* handles requests for multiple ports and, therefore, you can monitor several different Web Services.

### 3.3.9.2 Creating a Consumer with *Axis*

Creating consumers with *Axis* is really not that much different than the previous examples. There are some different include files and a tool to create proxy code much like the WSDL tool found in *.NET*, but the concept of creating consumers is the same.

**Command Line Application.** The following code is a simple command line application that invokes the *Axis* Web Service example shown earlier. The example starts off by defining all the different Java libraries needed for this example with the import statements. Then there is a class definition followed by the definition of the main method. Then the example begins to call the Web Service by reading the options passed in from the command line, creating a call to the URL where the Web Service resides, defining the class and method to call from the server, defining the result, and finally printing the result.

```
import org.apache.axis.AxisFault;
import org.apache.axis.client.Call;
import org.apache.axis.client.Service;
import org.apache.axis.encoding.XMLType;
import org.apache.axis.utils.Options;
```

```

import javax.xml.rpc.ParameterMode;
import javax.xml.namespace.QName;
import java.net.URL;

public class getSimpleStock {
    public static void main (String[] args)
    throws Exception {

        String symbol = null;
        //get command line args
        Options myOpts      = new Options( args );
        args = myOpts.getRemainingArgs();

        //begin a call to a Web Service
        Service myService = new Service();
        Call     myCall     = (Call) myService.createCall();

        //location of axis server
        myOpts.setDefaultURL("http://localhost:8000/axis/servlet/AxisServlet" );
        myCall.setTargetEndpointAddress( new URL(myOpts.getURL()) );
        myCall.setUseSOAPAction( true );
        //the method to call
        myCall.setSOAPActionURI( "getTestQuote" );
        //how to encode the request/response
        myCall.setEncodingStyle ("http://schemas.xmlsoap.org/soap/encoding/" );
        //define the class and method used
        myCall.setOperationName( new QName("SimpleStockExample", "getTestQuote") );
        //add symbol to the request
        myCall.addParameter ( "symbol", XMLType.XSD_STRING, ParameterMode.IN );
        myCall.setReturnType( XMLType.XSD_FLOAT );
        //make the actual object call
        Object myResult =
        myCall.invoke( new Object[] { symbol = args[0] } );
        //print the result
        System.out.println("This is the returned value: " +
        ((Float)myResult).floatValue());
    }
}

```

There are two approaches commonly adopted for defining and implementing web services, **Contract-first** and **Contract-last**. In the Contract-first approach, we start with a web service contract, which is a WSDL file. We use tools to generate java artifacts out of the WSDL file. These generated artifacts includes java interfaces and implementation classes as well as any other web services plumbing related code. Whereas in the Contract-last approach, you start with the Java code, and let the WSDL be generated from that.

Even though, the approach to be adopted depends on many factors including the context in which you are defining your web services, the Contract-first approach is preferred in normal circumstances. But one practical difficulty in the Contract-first approach is that creating a WSDL is not a trivial process and hence we may not be able to do that easily without some special tool support. To deal with this difficulty, there is a mixed approach which we can follow, whose steps are as follows:

- Create the web service interface (Java interface) alone first
- Generate WSDL out of this interface
- Now follow the normal steps which you follow in the Contract-first approach

We will follow this mixed approach in our samples. Hence, you will be able to adopt the samples here to follow either the Contract-first or the Contract-last approach by doing simple changes to the build scripts.

To make calling a Web Service easier with Apache tools, *Axis* comes with *WSDL2Java* application that generates Java proxy code to make calling the Web Service easier and quicker.

**Using WSDL2Java to create a proxy.** With Axis, WSDL is generated automatically and can be seen by simply putting “wsdl” in the query string.

Consider the following URL: <http://localhost:8080/axis/services/SimpleStockExample?wsdl> This will display the WSDL for the SimpleStockExample Web Service shown earlier in this chapter.

To create a Java proxy for a client to use, execute *WSDL2Java* and send the URL of the SimpleStockQuote’s WSDL output. The command should look something like the following.

```
java org.apache.axis.wsdl.WSDL2Java  
http://localhost:8080/axis/services/SimpleStockExample?wsdl
```

This creates a directory called localhost. Within that directory you will find the following four files.

```
SimpleStockExample.java           SimpleStockExampleSoapBindingStub.java  
SimpleStockExampleService.java   SimpleStockExampleServiceLocator.java.
```

SimpleStockExample and SimpleStockExampleService are both interfaces that define the functionality that the other two Java files must implement. SimpleStockExampleServiceLocator implements SimpleStockExample Service and defines the location of the Web Service and methods. SimpleStockExampleSoapBindingStub.java implements SimpleStockExampleServiceLocator.java and handles the code related to the call.

The following is the code for SimpleStockExample.java.

```
/* SimpleStockExample.java  
 * This file was auto-generated from WSDL by the Apache Axis WSDL2Java emitter.  
 */  
  
package localhost;  
  
public interface SimpleStockExample extends  
    java.rmi.Remote {  
    public float getTestQuote(java.lang.String in0)  
        throws java.rmi.RemoteException;  
}
```

Notice how the code simply defines an interface, and notice how SimpleStockExampleServiceLocator.java utilizes this interface.

```
/* SimpleStockExampleServiceLocator.java  
 * This file was auto-generated from WSDL by the Apache Axis WSDL2Java emitter.  
 */  
  
package localhost;  
  
public class SimpleStockExampleServiceLocator extends  
    org.apache.axis.client.Service implements localhost.SimpleStockExampleService {  
  
    // Use to get a proxy class for SimpleStockExample  
    private final java.lang.String  
    SimpleStockExample_address =  
        "http://localhost:8080/axis/services/SimpleStockExample";  
  
    public String getSimpleStockExampleAddress() {  
        return SimpleStockExample_address;  
    }  
  
    public localhost.SimpleStockExample getSimpleStockExample() throws  
        javax.xml.rpc.ServiceException {  
        java.net.URL endpoint;  
        try {  
            endpoint = new java.net.URL (SimpleStockExample_address);  
        }
```

```

        }
    catch (java.net.MalformedURLException e) {
        return null;
    }
    return getSimpleStockExample(endpoint);
}

public localhost.SimpleStockExample getSimpleStockExample
(java.net.URL portAddress) throws javax.xml.rpc.ServiceException {
    try {
        return new
localhost.SimpleStockExampleSoapBindingStub
(portAddress, this);
    }
    catch (org.apache.axis.AxisFault e) {
        return null; // ???
    }
}

/* For the given interface, get the stub implementation. If this service
 * has no port for the given interface, then ServiceException is thrown.
 */
public java.rmi.Remote getPort(Class serviceEndpointInterface) throws
javax.xml.rpc.ServiceException {
    try {
        if (localhost.SimpleStockExample.class.isAssignableFrom
(serviceEndpointInterface)) {
            return new localhost.SimpleStockExampleSoapBindingStub
(new java.net.URL(SimpleStockExample_address), this);
        }
    }
    catch (Throwable t) {
        throw new javax.xml.rpc.ServiceException(t);
    }
    throw new javax.xml.rpc.ServiceException
("There is no stub implementation for the interface: " +
(serviceEndpointInterface == null ? "null" :
serviceEndpointInterface.getName()));
}
}

```

The previous Java code is mainly centered on finding the name of the service. The following code snippet is SimpleStockExampleService.java. This defines the interface for the actually getting values from the Web Service.

```

/* SimpleStockExampleService.java
 * This file was auto-generated from WSDL by the Apache Axis WSDL2Java emitter.
 */

package localhost;

public interface SimpleStockExampleService extends javax.xml.rpc.Service {
    public String getSimpleStockExampleAddress();

    public localhost.SimpleStockExample getSimpleStockExample() throws
        javax.xml.rpc.ServiceException;

    public localhost.SimpleStockExample getSimpleStockExample
        (java.net.URL portAddress) throws javax.xml.rpc.ServiceException;
}

```

Finally, SimpleStockExampleSoapBindingStub.java contains the code defined in the previous interface. This code focuses on creating the call and getting values back from the Web Service.

```

/* SimpleStockExampleSoapBindingStub.java
 * This file was auto-generated from WSDL by the Apache Axis WSDL2Java emitter.

```

```

*/
package localhost;

public class SimpleStockExampleSoapBindingStub extends
org.apache.axis.client.Stub implements
localhost.SimpleStockExample {
private java.util.Vector cachedSerClasses = new java.util.Vector();
private java.util.Vector cachedSerQNames = new java.util.Vector();
private java.util.Vector cachedSerFactories = new java.util.Vector();
private java.util.Vector cachedDeserFactories = new java.util.Vector();

public SimpleStockExampleSoapBindingStub() throws org.apache.axis.AxisFault {
    this(null);
}

public SimpleStockExampleSoapBindingStub (java.net.URL endpointURL,
    javax.xml.rpc.Service service) throws org.apache.axis.AxisFault {
    this(service);
    super.cachedEndpoint = endpointURL;
}

public SimpleStockExampleSoapBindingStub (javax.xml.rpc.Service service) throws
org.apache.axis.AxisFault {
    try {
        if (service == null) {
            super.service = new org.apache.axis.client.Service();
        } else {
            super.service = service;
        }
    }
    catch(java.lang.Exception t) {
        throw org.apache.axis.AxisFault.makeFault(t);
    }
}

private org.apache.axis.client.Call createCall() throws
java.rmi.RemoteException {
    try {
        org.apache.axis.client.Call call = (org.apache.axis.client.Call)
            super.service.createCall();
        if (super.maintainSessionSet) {
            call.setMaintainSession(super.maintainSession);
        }
        if (super.cachedUsername != null) {
            call.setUsername(super.cachedUsername);
        }
        if (super.cachedPassword != null) {
            call.setPassword(super.cachedPassword);
        }
        if (super.cachedEndpoint != null) {
            call.setTargetEndpointAddress(super.cachedEndpoint);
        }
        if (super.cachedTimeout != null) {
            call.setTimeout(super.cachedTimeout);
        }
        java.util.Enumeration keys =
super.cachedProperties.keys();
        while (keys.hasMoreElements()) {
            String key = (String) keys.nextElement();
            if(call.isPropertySupported(key))
                call.setProperty(key, super.cachedProperties.get(key));
            else
                call.setScopedProperty(key, super.cachedProperties.get(key));
        }
        // All the type mapping information is registered when the first call
    }
}

```

```

        // is made. The type mapping information is actually registered in
        // the TypeMappingRegistry of the service, which
        // is the reason why registration is only needed for the first call.
        synchronized (this) {
            if (firstCall()) {
                // must set encoding style before registering serializers
                call.setEncodingStyle
                    (org.apache.axis.Constants.URI_SOAP11_ENC);
                for (int i = 0; i < cachedSerFactories.size(); ++i) {
                    Class cls = (Class) cachedSerClasses.get(i);
                    javax.xml.namespace.QName qName = (javax.xml.namespace.QName)
                        cachedSerQNames.get(i);
                    Class sf = (Class) cachedSerFactories.get(i);
                    Class df = (Class) cachedDeserFactories.get(i);
                    call.registerTypeMapping (cls, qName, sf, df, false);
                }
            }
            return call;
        }
    catch (Throwable t) {
        throw new org.apache.axis.AxisFault
        ("Failure trying to get the Call object", t);
    }
}
...
}

```

The last example code was cut short for a very good reason. You don't really need to know what the code is doing.

Once the Java files are created, compile them with the following command within the localhost directory:

```
javac *.java
```

This should make all the corresponding class files and compile without error. Now the classes are available to any client you wish to create. The following example utilizes these classes.

The import statement brings in all the classes that *WSDL2Java* created and you compiled. Then the class *getSimpleStockWSDL* is created with a main method. The result variable is defined so it handles a value returned by the method. Next, the code creates the object *myService*, which contains information about where the object and method you need to call reside. The next step creates the object *mySOAP*, which is the actual object that represents the *SimpleStockQuote* class. Notice that the next step is the *mySOAP* object calling the *getTestQuote* method. Then the value of *result* is output. This piece of code is the simplest call to a Java Web Service shown in this book so far.

```

import localhost.*;

public class getSimpleStockWSDL {
    public static void main(String [] args) throws
    Exception {
        //The type the service returns.
        double result;
        SimpleStockExampleServiceLocator myService =
            new SimpleStockExampleServiceLocator();
        localhost.SimpleStockExample mySOAP = myService.getSimpleStockExample();
        result = mySOAP.getTestQuote("C");
        System.out.println("This is the value: " + result);
    }
}

```

### 3.3.9.3 A Hello Example

We will implement the server in a Contract-first approach, but since we don't want to hand code the WSDL, let's start with a Java interface.

**IHelloWeb** is a simple Java interface, which defines a business method as shown here:

```
public interface IHelloWeb{
    public String hello(String param);
}
```

In the Contract-first approach, we start from a WSDL. As WSDL is language and platform neutral, we are sure that the client and server implemented in the Contract-first approach will be able to interoperate. But in our sample, we can start with a Java interface and then generate WSDL. So, in order to make sure that this generated WSDL is also compliant to interoperable standards, you need to pay attention to the parameters and return types of the method declaration in the java interface. Before you generate the WSDL make sure that the types in this Java interface can be interpreted as standard, portable types in the WSDL too.

Now we need to implement the web service. **HelloWebService** class will just do that.

```
public class HelloWebService implements IHelloWeb{
    private static int times;
    public HelloWebService(){
        System.out.println("Inside HelloWebService.HelloWebService..."); }
    public String hello(String param){
        System.out.println("Inside HelloWebService.hello... - " + (++times));
        return "Return From Server";
    }
}
```

Now, instead of creating the web service implementation class from scratch, we can generate an implementation template class. Into this template, you can manually add your business logic. OK, that is the method for your production deployments, but for this sample, you don't need to do these manual steps. Instead, we will try to do everything automatic using a smart ant build file.

The **build.xml** file is important, since it takes you step by step, starting from a Java interface through implementing business logic, and then packaging as a standard web archive. So, we will reproduce the entire build file here.

```
<?xml version="1.0" ?>
<project default="all">
    <property file="../examples.properties"/>
    <property name="build" value="build"/>
    <property name="dist" value="dist"/>
    <property name="lib" value="lib"/>
    <property name="src" value="src"/>
    <property name="gensrc" value="gensrc"/>
    <property name="config" value="config"/>
    <property name="webapp.name" value="AxisEndToEnd"/>
    <property name="service.name" value="HelloWebService"/>
    <property name="wsdl" value="HelloWebService.wsdl"/>
    <property name="interface.package"
              value="com.binildas.apache.axis.AxisEndToEnd"/>
    <property name="interface.path"
              value="com/binildas/apache/axis/AxisEndToEnd"/>
    <property name="interface.class" value="IHelloWeb"/>
    <property name="implement.package"
              value="com.binildas.apache.axis.AxisEndToEnd"/>
    <property name="implement.path"
              value="com/binildas/apache/axis/AxisEndToEnd"/>
    <property name="implement.class" value="HelloWebService"/>
```

```

<path id="classpath">
    <pathelement path=".build"/>
    <fileset dir="${axis.home}/lib">
        <include name="*.jar"/>
    </fileset>
</path>
<target name="all" depends="deploy, compileclient">
</target>
<target name="clean">
    <delete dir="${build}"/>
    <delete dir="${dist}"/>
    <delete dir="${lib}"/>
    <delete dir="${gensrc}"/>
</target>
<target name="init">
    <mkdir dir="${build}"/>
    <mkdir dir="${dist}"/>
    <mkdir dir="${lib}"/>
    <mkdir dir="${gensrc}"/>
</target>
<target name="copy">
    <copy todir="${lib}">
        <fileset dir="${axis.home}/lib">
            <include name="*.jar"/>
        </fileset>
    </copy>
</target>
<target name="precompile" depends="clean, init">
    <javac srcdir="${src}" destdir="build" classpathref="classpath">
        <exclude name="**/*Client*.java"/>
    </javac>
</target>
<target name="java2wsdl" depends="precompile">
    <java classname="org.apache.axis.wsdl.Java2WSDL" fork="true"
        failonerror="true">
        <arg value="-o"/>
        <arg value="${wsdl}"/>
        <arg value="-lhttp://localhost:8080/${webapp.name}/
            services/${service.name}"/>
        <arg value="${interface.package}.${interface.class}"/>
        <classpath>
            <path refid="classpath"/>
            <pathelement location="${build}"/>
        </classpath>
    </java>
</target>
<target name="wsdl2java" depends="java2wsdl">
    <java classname="org.apache.axis.wsdl.WSDL2Java" fork="true"
        failonerror="true">
        <arg value="-o"/>
        <arg value="${gensrc}"/>
        <arg value="-s"/>
        <arg value="-S"/>
        <arg value="no"/>
        <arg value="-c"/>
        <arg value="${implement.package}.${implement.class}"/>
        <arg value="${wsdl}"/>
        <classpath>
            <path refid="classpath"/>
            <pathelement location="${build}"/>
        </classpath>
    </java>
</target>

```

```

        </java>
    </target>
    <target name="implement" depends="wsdl2java">
        <delete>
            <fileset dir="${gensrc}/${implement.path}"
                includes="${implement.class}.java"/>
        </delete>
        <copy todir="${gensrc}/${implement.path}" overwrite="ture">
            <fileset dir="${src}/${implement.path}">
                <include name="${implement.class}.java"/>
            </fileset>
        </copy>
    </target>
    <target name="compile" depends="implement">
        <javac srcdir="${gensrc}" destdir="build"
            classpathref="classpath"/>
    </target>
    <target name="compileclient">
        <javac srcdir="${src}" destdir="build" classpathref="classpath">
            <include name="**/*Client*.java"/>
        </javac>
    </target>
    <target name="deploy" depends="compile, copy">
        <move todir="${config}" flatten="yes">
            <fileset dir="${gensrc}">
                <include name="**/*.wsdd"/>
            </fileset>
        </move>
        <java classname="org.apache.axis.utils.Admin" fork="true"
            failonerror="true" dir="config">
            <arg value="server"/>
            <arg file="config/deploy.wsdd" />
            <classpath>
                <path refid="classpath"/>
                <pathelement location="build"/>
            </classpath>
        </java>
        <war destfile="dist/${webapp.name}.war" webxml="config/web.xml">
            <webinf dir="config">
                <include name="server-config.wsdd"/>
            </webinf>
            <lib dir="lib"/>
            <classes dir="build"/>
        </war>
        <delete dir="${lib}"/>
    </target>
</project>

```

Let us now understand the implementation of the web service step by step. We will execute the following ant targets, in the same order.

- clean: This will delete all temporary folders and any generated files in the previous build.
- init: This will create a few, new folders.
- precompile: In this step, the aim is to compile the interface class.
- java2wsdl: The java2wsdl will generate the WSDL from the precompiled java interface. You can look at the URL <http://ws.apache.org/axis/java/reference.html> to get an understanding of the options available in this step.
- wsdl2java: Now, we start our Contract-first process. As we have the WSDL, we will use wsdl2java tools to create web service artifacts including the implementation template class. These generated files are placed in a folder, for example, gensrc. This step will also generate deploy.wsdd and undeploy.wsdd, two files which will

help us generate server side deployment configurations later.

- implement: As mentioned previously, to avoid the manual process of adding code to the generated implementation template, we already have a Java file with the same name as the generated implementation template, which contains the same Java class, HelloWebService, with the business code implemented. So we will overwrite the generated file with the file already present, which in effect is equivalent to adding business code to the generated file.
- compile: We will now compile all the generated files, including the web service implementation class containing the business logic.
- copy: This will bring all required axis libraries to a staging directory for example, lib in our codebase, so that it is easy to package them into the web archive.
- deploy: In this step, we will use the org.apache.axis.utils.Admin class to generate the deployment configuration file, server-config.wsdd, taking deploy.wsdd as the input. We then create a standard web archive, which can be readily deployed into your favorite web server.
- compileclient: As a last step, we will compile the Client code too.

**RpcClient** makes use of auto generated client side stub classes to invoke the remote web service in an RPC style.

```
public class RpcClient{  
    private static String wsdlUrl = "http://localhost:8080/  
                                     AxisEndToEnd/services/HelloWebService?WSDL";  
    private static String namespaceURI = "http://AxisEndToEnd.axis.  
                                         apache.binildas.com";  
    private static String localPart = "IHelloWebService";  
    protected void executeClient(String[] args) throws Exception{  
        IHelloWebService iHelloWebService = null;  
        IHelloWeb iHelloWeb = null;  
        if(args.length == 3){  
            iHelloWebService = new IHelloWebServiceLocator(args[0],  
                                              new QName(args[1], args[2]));}  
        else{  
            iHelloWebService = new IHelloWebServiceLocator(wsdlUrl,  
                                              new QName(namespaceURI, localPart));}  
        iHelloWeb = iHelloWebService.getHelloWebService();  
        log("Response From Server : " + iHelloWeb.hello("Binil"));  
    }  
    public static void main(String[] args) throws Exception{  
        RpcClient client = new RpcClient();  
        client.executeClient(args);  
    } }
```

We have provided one more client code called **CallClient**, which will use Axis and SOAP APIs to invoke the web service in a document oriented manner.

```
public class CallClient {  
    public static String wsURL =  
        "http://localhost:8080/AxisEndToEnd/services/HelloWebService?WSDL";  
    public static String action = "HelloWebService";  
    //SOAP Request - Not shown fully  
    public static String msg = "<?xml version=\"1.0\""  
        encoding=\"UTF-8\"?><soapenv:Envelope ...>";  
    public static void test() throws Exception{  
        InputStream input = new ByteArrayInputStream(msg.getBytes());  
        Service service = new Service();  
        Call call = (Call) service.createCall();  
        SOAPEnvelope soapEnvelope = new SOAPEnvelope(input);  
        call.setTargetEndpointAddress( new URL(wsURL) );  
        if (action != null) {  
            call.setUseSOAPAction( true );  
            call.setSOAPActionURI( action );  
        } }
```

```

        soapEnvelope = call.invoke( soapEnvelope );
        System.out.println( "Response:\n" + soapEnvelope.toString() );
    }
    public static void main(String args[]) throws Exception{
        CallClient callClient = new CallClient();
        if(args.length > 0){ wsURL = args[0]; }
        if(args.length > 1){ action = args[1]; }
        callClient.test(); }
}

```

The document oriented web service request has not been fully shown in the code. But you can look at the source code to view it fully.

The previous build.xml was a bit lengthy, and we again agree that the 10 steps mentioned earlier to implement the web service are not trivial ones for a novice user to execute. But believe it; we are going to do all those things with just one ant command. So, save the above build.xml file so that you can re-use them in your projects too. To build the server side code, execute ant. At the end of the build, we will have the deployable web archive (AxisEndToEnd.war).

You can now transfer this archive to the webapps folder of your web server and restart your server. Assuming the deployment went fine, the WSDL for the web service will be available now at the URL

<http://localhost:8080/AxisEndToEnd/services>HelloWebService?WSDL>.

You can now execute the client code to test your web service. Since we have provided two versions of client code, there are two options for you to test the web service. To execute the RpcClient, execute the following command: ant runrpc. To execute the CallClient, execute the following command: ant runcall.

### 3.3.9.4 Overview of Axis2

This tool is part of "Web Service Project @ Apache", a set of several projects that cover various aspects related to the development and usage of web services. They are for the majority implementations of protocols and specifications.

Axis2 is a complete re-factoring of the previous 1.x version, and upon its architecture two different implementations were built, Axis2/Java and Axis2/C. The SAX event-based XML parser used in the previous version has been replaced by the pull-based StAX, which allows greater control over the document processing that translates into higher efficiency and performances. Indeed Axis2 uses AXIOM (AXIIs Object Model), a light-weight object model for XML processing, which is based on StAX and offers enhancing features. Axis2 can be used to develop both SOAP-based and RESTful web services. Also, it supports Asynchronous (or Non-Blocking) web services invocation, which is implemented by Callback mechanisms.

It has been designed with a modular and extensible architecture. The processes of sending and receiving the SOAP messages are performed by two "Pipes" ("Flows"): In Pipe and Out Pipe. Each pipe is designed to process the message throughout a sequence of phases which can have pluggable "Handlers", thereby giving the whole architecture a high-level of extensibility. In addition to built-in phases, there is the option to add User-defined phases with custom handlers inside them, in order to perform new mechanisms or to override existing ones.

Another important level of Handlers grouping is the concept of Module. Each module defines a set of handlers, and a descriptor of the phase rules. A handler can specify not only the phase where it will perform its action, but also its execution order inside the phase throughout the phase rules. In the Axis2 language, a module is "available" when it is present in the system, though not active. The activation of a module turns it to the "engaged" state, after which its handlers are placed into their associated phases and enabled. Thus, it is easy, for example, to plug-in modules that handle WS-\* specifications such as WS-Security (Apache Rampart module) or WS-Atomic Transaction (Apache Kandula2 module).

The data binding is not part of the core of Axis2, but it is provided by an extension mechanism that allows the choice among ABD (Axis Data Binding), XMLBeans, JAX-Me, and JibX. Also, the range of the transmission protocols supported is complete (HTTP, TCP, SMTP, and JMS).

The services, modules, phases and handlers are configured in an independent way with respect to the other deployed web applications. If you have a set of deployed web applications, each with its own business layer, you will not put a service into one of these web applications. Instead, it will be defined and configured into axis2 application and therefore generally available to every consumer (a client or another server application).

### 3.3.10 Web Services Support in J2EE

The Java 2 Platform Enterprise Edition (J2EE) is one of the two primary platforms currently being used to develop enterprise solutions using Web services.

Starting with the J2EE 1.4 platform, with its main focus on Web services, the existing Java-XML technologies are integrated into a consolidated platform in a standard way, thereby allowing applications to be exposed as Web services through a SOAP/HTTP interface. The next sections briefly describe the Web service-specific additions made in the J2EE 1.4 platform.

#### 3.3.10.1 Platform Overview

The Java 2 Platform is divided into three major development and runtime platforms, each addressing a different type of solution. The Java 2 Platform Standard Edition (J2SE) is designed to support the creation of desktop applications, while the Micro Edition (J2ME) is geared toward applications that run on mobile devices. The Java 2 Platform Enterprise Edition (J2EE) is built to support large-scale, distributed solutions. J2EE has been in existence for over five years and has been used extensively to build traditional n-tier applications with and without Web technologies.

The J2EE development platform consists of numerous composable pieces that can be assembled into full-fledged Web solutions. Let's take a look at some of the technologies more relevant to Web services.

The Servlets + EJBs and Web + EJB Container layers (as well as the JAX-RPC Runtime) relate to the Web and Component Technology layers

There are many J2EE standards published by Sun Microsystems that establish the parts of the J2EE architecture to which vendors that implement and build products around this environment must conform. Three of the more significant specifications that pertain to SOA are listed here:

- Java 2 Platform Enterprise Edition Specification. This important specification establishes the distributed J2EE component architecture and provides foundation standards that J2EE product vendors are required to fulfill in order to claim J2EE compliance.
- Java API for XML-based RPC (JAX-RPC). This document defines the JAX-RPC environment and associated core APIs. It also establishes the Service Endpoint Model used to realize the JAX-RPC Service Endpoint, one of the primary types of J2EE Web services.
- Web Services for J2EE. The specification that defines the vanilla J2EE service architecture and clearly lays out what parts of the service environment can be built by the developer, implemented in a vendor-specific manner, and which parts must be delivered according to J2EE standards.

**Architecture components.** J2EE solutions inherently are distributed and therefore componentized. The following types of components can be used to build J2EE Web applications:

- Java Server Pages (JSPs). Dynamically generated Web pages hosted by the Web server. JSPs exist as text files comprised of code interspersed with HTML.
- Struts. An extension to J2EE that allows for the development of Web applications with sophisticated user-interfaces and navigation.
- Java Servlets. These components also reside on the Web server and are used to process HTTP request and response exchanges. Unlike JSPs, servlets are compiled programs.
- Enterprise JavaBeans (EJBs). The business components that perform the bulk of the processing within enterprise solution environments. They are deployed on dedicated application servers and can therefore leverage middleware features, such as transaction support.

While the first two components are of more relevance to establishing the presentation layer of a service-oriented solution, the latter two commonly are used to realize Web services.

**Runtime environment.** The J2EE environment relies on a foundation Java runtime to process the core Java parts of any J2EE solution. In support of Web services, J2EE provides additional runtime layers that, in turn, supply additional Web services specific APIs (explained later). Most notable is the JAX-RPC runtime, which establishes fundamental services, including support for SOAP communication and WSDL processing.

Additionally, implementations of J2EE supply two types of component containers that provide hosting environments geared toward Web services-centric applications that are generally EJB or servlet-based.

- EJB container. This container is designed specifically to host EJB components, and it provides a series of enterprise-level services that can be used collectively by EJBs participating in the distributed execution of a business task. Examples of these services include transaction management, concurrency management, operation-level security, and object pooling.
- Web container. A Web container can be considered an extension to a Web server and is used to host Java Web applications consisting of JSP or Java servlet components. Web containers provide runtime services geared toward the processing of JSP requests and servlet instances.

EJB and Web containers can host EJB-based or servlet-based J2EE Web services. Web service execution on both containers is supported by JAX-RPC runtime services. However, it is the vendor-specific container logic that generally determines the shape and form of the system-level message processing logic provided in support of Web services.

J2EE vendors provide containers as part of their server products. A container then establishes the runtime that hosts an instance of the vendor's server software. Examples of currently available containers are Sun ONE (Open Network Environment) Application Server, IBM WebSphere, and Oracle Application Server Containers for J2EE (OC4J).

**Programming languages.** The Java 2 Platform Enterprise Edition is centered around the Java programming language. Different vendors offer proprietary development products that provide an environment in which the standard Java language can be used to build Web services. Examples of currently available development tools are Rational Application Developer from IBM, Java Studio from Sun Microsystems, and JDeveloper from Oracle.

### 3.3.10.2 APIs

J2EE contains several APIs for programming functions in support of Web services. The classes that support these APIs are organized into a series of packages. Here are some of the APIs relevant to building SOA.

- **Java API for XML Processing (JAXP)** This API is used to process XML document content using a number of available parsers. Both Document Object Model (DOM) and Simple API for XML (SAX) compliant models are supported, as well as the ability to transform and validate XML documents using XSLT stylesheets and XSD schemas. Example packages include:
  - javax.xml.parsers A package containing classes for different vendor-specific DOM and SAX parsers
  - org.w3c.dom and org.xml.sax These packages expose the industry standard DOM and SAX document models.
  - javax.xml.transform A package providing classes that expose XSLT transformation functions.
- **Java API for XML-based RPC (JAX-RPC)** .The most established and popular SOAP processing API, supporting both RPC-literal and document-literal request-response exchanges and one-way transmissions. Example packages that support this API include:
  - javax.xml.rpc and javax.xml.rpc.server These packages contain a series of core functions for the JAX-RPC API.
  - javax.xml.rpc.handler and javax.xml.rpc.handler.soap API functions for runtime message handlers are provided by these collections of classes.
  - javax.xml.soap and javax.xml.rpc.soapAPI functions for processing SOAP message content and bindings.
- **Java API for XML Registries (JAXR)**. An API that offers a standard interface for accessing business and service registries. Originally developed for ebXML directories, JAXR now includes support for UDDI.
  - javax.xml.registry A series of registry access functions that support the JAXR API.
  - javax.xml.registry.infomodel Classes that represent objects within a registry.
- **Java API for XML Messaging (JAXM)**. An asynchronous, document-style SOAP messaging API that can be used for one-way and broadcast message transmissions (but can still facilitate synchronous exchanges as well).

- **SOAP with Attachments API for Java (SAAJ).** Provides an API specifically for managing SOAP messages requiring attachments. The SAAJ API is an implementation of the SOAP with Attachments (SwA) specification.
- **Java Architecture for XML Binding API (JAXB).** This API provides a means of generating Java classes from XSD schemas and further abstracting XML-level development.
- **Java Message Service API (JMS).** A Java-centric messaging protocol used for traditional messaging middleware solutions and providing reliable delivery features not found in typical HTTP communication.

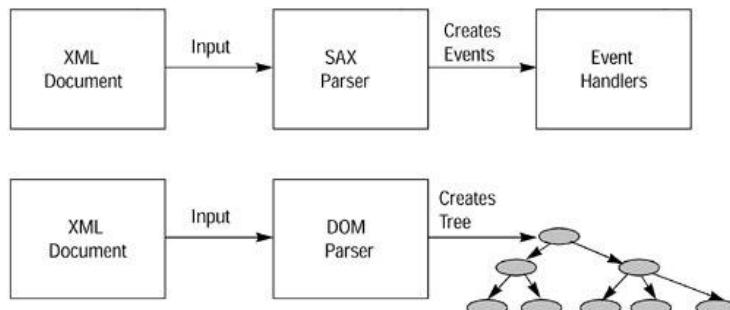
Of these APIs, the two most commonly used for SOA are JAX-RPC to govern SOAP messaging and JAXP for XML document processing. The two other packages relevant to building the business logic for J2EE Web services are javax.ejb and javax.servlet, which provide fundamental APIs for the development of EJBs and servlets.

### 3.3.10.3 Java APIs for XML Processing (JAXP)

Java APIs for XML Processing (JAXP) is a vendor-neutral set of lightweight APIs for parsing or processing XML documents. Because XML is the common language enabling Web services, an XML parser is a necessity to process the messages—the XML documents—exchanged among Web services. JAXP processes XML documents using the SAX or DOM models, and it permits use of XSLT engines during the document processing. (XSLT, which stands for eXtensible Stylesheet Language Transformation, is used for transforming XML documents from one format to another.)

The main JAXP APIs are available through the `javax.xml.parsers` package, which provides two vendor-agnostic factory interfaces—one interface for SAX processing and another for DOM processing. These factory interfaces allow the use of other JAXP implementations.

Figure 3.33 shows how the SAX and DOM parsers function. SAX processes documents serially, converting the elements of an XML document into a series of events. Each particular element generates one event, with unique events representing various parts of the document. User-supplied event handlers handle the events and take appropriate actions. SAX processing is fast because of its serial access and small memory storage requirements.



**Fig. 3.33. SAX- and DOM-Based XML Parser APIs**

Code Example 3.33 shows how to use the JAXP APIs and SAX to process an XML document.

#### Example 3.33. Using SAX to Process an XML Document

```

public class AnAppThatUsesSAXForXMLProcessing extends DefaultHandler {
    public void someMethodWhichReadsXMLDocument() {
        // Get a SAX Parser Factory and set validation to true
        SAXParserFactory spf = SAXParserFactory.newInstance();
        spf.setValidating(true);
        // Create a JAXP SAXParser
        SAXParser saxParser = spf.newSAXParser();
        // Get the encapsulated SAX XMLReader
        xmlReader = saxParser.getXMLReader();
        // Set the ContentHandler of the XMLReader
        xmlReader.setContentHandler(this);
        // Tell the XMLReader to parse the XML document
        xmlReader.parse(XMLDocumentName);    }   }

```

DOM processing creates a tree from the elements in the XML document. Although this requires more memory (to store the tree), this feature allows random access to document content and enables splitting of documents into fragments, which makes it easier to code DOM processing. DOM facilitates creations, changes, or additions to incoming XML documents. Code Example DES.E8 shows how to use the JAXP APIs and DOM to process an XML document.

#### **Example 3.34. Using DOM to Process an XML Document**

```
public class AnAppThatUsesDOMForXMLProcessing {
    public void someMethodWhichReadsXMLDocument() {
        // Step 1: create a DocumentBuilderFactory
        DocumentBuilderFactory dbf = DocumentBuilderFactory.newInstance();
        dbf.setValidating(true);
        // Step 2: create a DocumentBuilder that satisfies
        // the constraints specified by the DocumentBuilderFactory
        db = dbf.newDocumentBuilder();
        // Step 3: parse the input file
        Document doc = db.parse(XMLDocumentFile);
        // Parse the tree created - node by node
    }
}
```

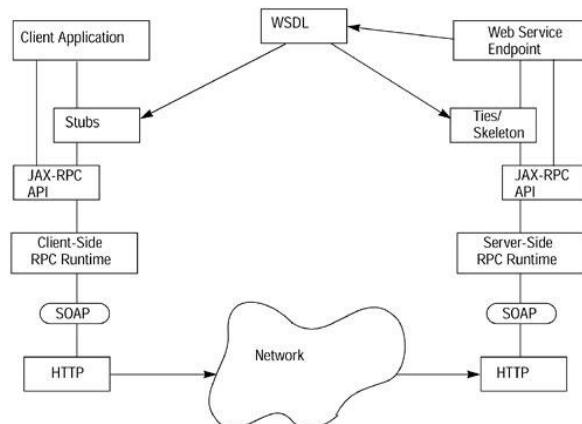
#### **3.3.10.4 Java API for XML-Based RPC (JAX-RPC)**

Java API for XML-based RPC (JAX-RPC) supports XML-based RPC for Java and J2EE platforms. It enables a traditional client-server remote procedure call (RPC) mechanism using an XML-based protocol. JAX-RPC enables Java technology developers to develop SOAP-based interoperable and portable Web services. Developers use the JAX-RPC programming model to develop SOAP-based Web service endpoints, along with their corresponding WSDL descriptions, and clients. A JAX-RPC-based Web service implementation can interact with clients that are not based on Java. Similarly, a JAX-RPC-based client can interact with a non-Java-based Web service implementation.

For typical Web service scenarios, using JAX-RPC reduces complexity for developers by:

- Standardizing the creation of SOAP requests and responses
- Standardizing marshalling and unmarshalling of parameters and other runtime and deployment-specific details
- Removing these SOAP creation and marshalling/unmarshalling tasks from a developer's responsibilities by providing these functions in a library or a tool
- Providing standardized support for different mapping scenarios, including XML to Java, Java to XML, WSDL-to-Java, and Java-to-WSDL mappings

JAX-RPC also defines standard mappings between WSDL/XML and Java, which enables it to support a rich type set. However, developers may use types that do not have standard type mappings. JAX-RPC defines a set of APIs for an extensible type mapping framework that developers can use for types with no standard type mappings. With these APIs, it is possible to develop and implement pluggable serializers and de-serializers for an extensible mapping. Figure 3.34 shows the high-level architecture of the JAX-RPC implementation.



**Fig. 3.34. JAX-RPC Architecture**

A client application can make a request to a Web service in one of three ways:

1. Invoking methods on generated stubs— Based on the contents of a WSDL description of a service, tools can be used to generate stubs. These generated stubs are configured with all necessary information about the Web service and its endpoint. The client application uses the stubs to invoke remote methods available in the Web service endpoint.
2. Using a dynamic proxy— A dynamic proxy supports a Web service endpoint. When this mode is used, there is no need to create endpoint-specific stubs for the client.
3. Using a dynamic invocation interface (DII)— In this mode, operations on target service endpoints are accessed dynamically based on an in-memory model of the WSDL description of the service.

No matter which mode is used, the client application's request passes through the client-side JAX-RPC runtime. The runtime maps the request's Java types to XML and forms a corresponding SOAP message for the request. It then sends the SOAP message across the network to the server.

On the server side, the JAX-RPC runtime receives the SOAP message for the request. The server-side runtime applies the XML to Java mappings, then maps the request to the corresponding Java method call, along with its parameters.

Note that a client of a JAX-RPC service may be a non-Java client. Also, JAX-RPC can interoperate with any Web service, whether that service is based on JAX-RPC or not. Also note that developers need only deal with JAX-RPC APIs; all the details for handling SOAP happen under the hood.

JAX-RPC supports three modes of operation:

1. Synchronous request-response mode— After a remote method is invoked, the service client's thread blocks until a return value or exception is returned.
2. One-way RPC mode— After a remote method is invoked, the client's thread is not blocked and continues processing. No return value or exception is expected on this call.
3. Non-blocking RPC invocation mode— A client invokes a remote procedure and continues in its thread without blocking. Later, the client processes the remote method return by performing a blocked receive call or by polling for the return value.

In addition, JAX-RPC, by specifying a standard way to plug in SOAP message handlers, allows both pre- and post-processing of SOAP requests and responses. These message handlers can intercept incoming SOAP requests and outgoing SOAP responses, allowing the service to do additional processing.

Code Example 3.35 is an example of a JAX-RPC service interface for a simple service that provides weather information for a city.

#### ***Example 3.35. JAX-RPC Service Endpoint Interface Example***

```
public interface WeatherService extends Remote {  
    public String getWeather(String city) throws RemoteException;  
}
```

Code Example 3.36 shows the implementation of the weather service interface using a Web component.

#### ***Example 3.36. JAX-RPC Service Implementation***

```
public class WeatherServiceImpl implements WeatherService, ServiceLifecycle {  
    public void init(Object context) throws JAXRPCException {}  
    public String getWeather(String city) {  
        return ("Early morning fog clearing midday; " +  
               "over all great day expected in " + city);  
    }  
    public void destroy() {}  
}
```

Code Example 3.37 shows how a client, using JAX-RPC to access this weather service.

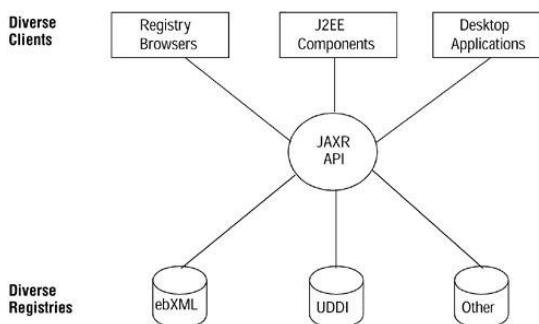
### **Example 3.37. A Java/J2EE Client Accessing the Weather Service**

```
....  
Context ic = new InitialContext();  
Service svc = (Service) ic.lookup("java:comp/env/service/WeatherService");  
WeatherSvcIntf port = (WeatherSvcIntf) svc.getPort(WeatherSvcIntf.class);  
String info = port.getWeather("New York");  
....
```

These examples illustrate that a developer has to code very little configuration and deployment information. The JAX-RPC implementation handles the details of creating a SOAP request, handling the SOAP response, and so forth, thereby relieving the developer of these complexities.

### **3.3.10.5 Java API for XML Registries (JAXR)**

Java API for XML Registries (JAXR), a Java API for accessing business registries, has a flexible architecture that supports UDDI, and other registry specifications (such as ebXML). Figure 3.35 illustrates the JAXR architecture.



**Fig. 3.35. JAXR Architecture**

A JAXR client, which can be a stand-alone Java application or a J2EE component, uses an implementation of the JAXR API provided by a JAXR provider to access business registries. A JAXR provider consists of two parts: a registry-specific JAXR provider, which provides a registry-specific implementation of the API, and a JAXR pluggable provider, which implements those features of the API that are independent of the type of registry. The pluggable provider hides the details of registry-specific providers from clients.

The registry-specific provider plugs into the pluggable provider, and acts on requests and responses between the client and the target registry. The registry-specific provider converts client requests into a form understood by the target registry and sends the requests to the registry provider using registry-specific protocols. It converts responses from the registry provider from a registry-specific format to a JAXR response, then passes the response to the client.

### **3.3.10.6 SOAP with Attachments API for Java (SAAJ)**

SOAP with Attachments API for Java (SAAJ), which enables developers to produce and consume messages conforming to the SOAP specification and SOAP with Attachments note, provides an abstraction for handling SOAP messages with attachments. Advanced developers can use SAAJ to have their applications operate directly with SOAP messages. Attachments may be complete XML documents, XML fragments, or MIME-type attachments. In addition, SAAJ allows developers to enable support for other MIME types. JAX technologies, such as JAX-RPC, internally use SAAJ to hide SOAP complexities from developers.

SAAJ allows the following modes of message exchanges:

- Synchronous request-response messaging—the client sends a message and then waits for the response
- One-way asynchronous messaging (also called fire and forget)—the client sends a message and continues with its processing without waiting for a response

### 3.3.10.7 Web Service Technologies Integrated in J2EE Platform

Not only are the Java XML technologies integrated into the platform, the platform also defines Web service-related responsibilities for existing Web and EJB containers, artifacts, and port components. The J2EE 1.4 platform ensures portability by integrating the Java XML technologies as extensions to existing J2EE containers, packaging formats, deployment models, and runtime services.

A Web service on the J2EE platform may be implemented as follows:

- Using a JAX-RPC service endpoint—The service implementation is a Java class in the Web container. The service adheres to the Web container's servlet lifecycle and concurrency requirements.
- Using an EJB service endpoint—The service implementation is a stateless session bean in an EJB container. The service adheres to the EJB container's lifecycle and concurrency requirements.

In either case, the service is made portable with the definition of a port component, which provides the service's outside view for Web service implementation. A port component consists of:

- A WSDL document describing the Web service that its clients can use
- A service endpoint interface defining the Web service's methods that are available to clients
- A service implementation bean implementing the business logic of the methods defined in the service endpoint interface. The implementation may be either a Java class in the Web container or a stateless session bean in the EJB container.

Container-specific service interfaces, created by the J2EE container, provide static stub and dynamic proxies for all ports. A client of a J2EE platform Web service can be a Web service peer, a J2EE component, or a stand-alone application. It is not required that the client be a Web service or application implemented in Java.

How do clients use a J2EE platform Web service? Here is an example of a J2EE component that is a client of some Web service. Such a client uses JNDI to look up the service, then it accesses the Web service's port using methods defined in the javax.xml.rpc.Service interface. The client accesses the service's functionality using its service endpoint interface. A client that is a J2EE component needs only consider that the Web service implementation is stateless. Thus, the client cannot depend on the service holding state between successive service invocations. A J2EE component client does not have to know any other details of the Web service, such as how the service interface accesses the service, the service implementation, how its stubs are generated, and so forth.

Recall what a Web service interface, such as the weather Web service, looks like when implemented as a JAX-RPC service endpoint on a J2EE platform. In contrast, Code Example 3.38 shows the equivalent EJB service endpoint implementation for the same weather service.

#### ***Example 3.38. EJB Service Endpoint Implementation for a Weather Service***

```
public class HelloService implements SessionBean {  
    private SessionContext sc;  
    public WeatherService() {}  
    public void ejbCreate() {}  
    public String getWeather(String city) {  
        return ("Early morning fog clearing midday; " +  
               "over all great day expected in " + city);  
    }  
    public void setSessionContext(SessionContext sc) {  
        this.sc = sc;  
    }  
    public void ejbRemove() {}  
    public void ejbActivate() {}  
    public void ejbPassivate() {}  
}
```

Any client can use the code shown in Code Example 3.37 to access this weather service. This holds true

- Regardless of whether the service is implemented as a JAX-RPC service endpoint or an EJB service endpoint
- Regardless of whether the client is a servlet, an enterprise bean, or a stand-alone Java client

**Service providers.** As previously mentioned, J2EE Web services are typically implemented as servlets or EJB components. Each option is suitable to meet different requirements but also results in different deployment configurations, as explained here:

- **JAX-RPC Service Endpoint.** When building Web services for use within a Web container, a JAX-RPC Service Endpoint is developed that frequently is implemented as a servlet by the underlying Web container logic. Servlets are a common incarnation of Web services within J2EE and most suitable for services not requiring the features of the EJB container.
- **EJB Service Endpoint.** The alternative is to expose an EJB as a Web service through an EJB Service Endpoint. This approach is appropriate when wanting to encapsulate existing legacy logic or when runtime features only available within an EJB container are required. To build an EJB Service Endpoint requires that the underlying EJB component be a specific type of EJB called a Stateless Session Bean.

Regardless of vendor platform, both types of J2EE Web services are dependent on the JAX-RPC runtime and associated APIs.

A frequent point of confusion is the naming of the JAX-RPC Service Endpoint and the JAX-RPC runtime. Many initially assume that the JAX-RPC runtime is associated only with the JAX-RPC Service Endpoint and the Web container. However, because JAX-RPC establishes a standardized service processing layer that spans both Web and EJB containers, its runtime applies to both JAX-RPC Service Endpoints and EJB Service Endpoints.

Also a key part of either service architecture is an underlying model that defines its implementation, called the Port Component Model. As described in the Web Services for J2EE specification, it establishes a series of components that comprise the implementation of a J2EE service provider, including:

- **Service Endpoint Interface (SEI).** A Java-based interpretation of the WSDL definition that is required to follow the JAX-RPC WSDL-to-Java mapping rules to ensure consistent representation.
- **Service Implementation Bean.** A class that is built by a developer to house the custom business logic of a Web service. The Service Implementation Bean can be implemented as an EJB Endpoint (Stateless Session Bean) or a JAX-RPC Endpoint (servlet). For an EJB Endpoint, it is referred to as an EJB Service Implementation Bean and therefore resides in the EJB container. For the JAX-RPC Endpoint, it is called a JAX-RPC Service Implementation Bean and is deployed in the Web container.

**Service requestors.** The JAX-RPC API also can be used to develop service requestors. It provides the ability to create three types of client proxies, as explained here:

- **Generated stub.** The generated stub (or just "stub") is the most common form of service client. It is auto-generated by the JAX-RPC compiler (at design time) by consuming the service provider WSDL, and producing a Java-equivalent proxy component. Specifically, the compiler creates a Java remote interface for every WSDL portType which exposes methods that mirror WSDL operations. It further creates a stub based on the WSDL port and binding constructs. The result is a proxy component that can be invoked as any other Java component. JAX-RPC takes care of translating communication between the proxy and the requesting business logic component into SOAP messages transmitted to and received from the service provider represented by the WSDL.
- **Dynamic proxy and dynamic invocation interface.** Two variations of the generated stub are also supported. The dynamic proxy is similar in concept, except that the actual stub is not created until its methods are invoked at runtime. Secondly, the dynamic invocation interface bypasses the need for a physical stub altogether and allows for fully dynamic interaction between a Java component and a WSDL definition at runtime.

The latter options are more suited for environments in which service interfaces are more likely to change or for which component interaction needs to be dynamically determined. For example, because a generated stub produces a static proxy interface, it can be rendered useless when the corresponding WSDL definition changes. Dynamic proxy generation avoids this situation.

**Service agents.** Vendor implementations of J2EE platforms often employ numerous service agents to perform a variety of runtime filtering, processing, and routing tasks. A common example is the use of service agents to process SOAP headers. To support SOAP header processing, the JAX-RPC API allows for the creation of specialized service agents called handlers runtime filters that exist as extensions to the J2EE container environments. Handlers can process SOAP header blocks for messages sent by J2EE service requestors or for messages received by EJB Endpoints and JAX-RPC Service Endpoints. Multiple handlers can be used to process different header blocks in the same SOAP message. In this case the handlers are chained in a predetermined sequence (appropriately called a handler chain).

**Platform extensions.** Different vendors that implement and build around the J2EE platform offer various platform extensions in the form of SDKs that extend their development tool offering. The technologies supported by these toolkits, when sufficiently mature, can further support contemporary SOA. Following are two examples of currently available platform extensions.

- IBM Emerging Technologies Toolkit. A collection of extensions that provide prototype implementations of a number of fundamental WS-\* extensions, including WS-Addressing, WS-ReliableMessaging, WS-MetadataExchange, and WS-Resource Framework.
- Java Web Services Developer Pack. A toolkit that includes both WS-\* support as well as the introduction of new Java APIs. Examples of the types of extensions provided include WS-Security (along with XML-Signature), and WS-I Attachments.

The WS-Resource Framework consists of a collection of specifications (WS-ResourceProperties, WS-ResourceLifetime, WS-BaseFaults, and WS-Service-Group) that establish a means of managing state information associated with Web services.

**Service encapsulation.** The distributed nature of the J2EE platform allows for the creation of independent units of processing logic through Enterprise Java Beans or servlets. EJBs or servlets can contain small or large amounts of application logic and can be composed so that individual units comprise the processing requirements of a specific business task or an entire solution. Both EJBs and servlets can be encapsulated using Web services. This turns them into EJB and JAX-RPC Service Endpoints, respectively. The underlying business logic of an endpoint can further compose and interact with non-endpoint EJB and servlet components. As a result, well-defined services can be created in support of SOA.

**Loose coupling.** The use of interfaces within the J2EE platform allows for the abstraction of metadata from a component's actual logic. When complemented with an open or proprietary messaging technology, loose coupling can be realized. EJB and JAX-RPC Endpoints further establish a standard WSDL definition, supported by J2EE HTTP and SOAP runtime services. Therefore, loose coupling is a characteristic that can be achieved in support of SOA.

**Messaging.** Prior to the acceptance of Web services, the J2EE platform supported messaging via the JMS standard, allowing for the exchange of messages between both servlets and EJB components. With the arrival of Web services support, the JAX-RPC API provides the means of enabling SOAP messaging over HTTP. Also worth noting is the availability of the SOAP over JMS extension, which supports the delivery of SOAP messages via the JMS protocol as an alternative to HTTP. The primary benefit here is that this approach to data exchange leverages the reliability features provided by the JMS framework. Within SOA this extension can be used by the business logic of a Web service, allowing SOAP messages to be passed through from the message process logic (which generally will rely on HTTP as the transport protocol). Either way, the J2EE platform provides the required messaging support for primitive SOA.

**Autonomy.** For a service to be fully autonomous, it must be able to independently govern the processing of its underlying application logic. A high level of autonomy is more easily achieved when building Web services that do not need to encapsulate legacy logic. JAX-RPC Service Endpoints exist as standalone servlets deployed within the Web container and are generally built in support of newer SOA environments. It may therefore be easier for JAX-RPC Service Endpoints to retain complete autonomy, especially when they are only required to execute a small amount of business logic. EJB Service Endpoints are required to exist as Stateless Session Beans, which supports autonomy within the immediate endpoint logic. However, because EJB Service Endpoints are more likely to represent existing legacy logic (or a combination of new and legacy EJB components), retaining a high level of autonomy can be challenging.

**Reusability.** The advent of Enterprise Java Beans during the rise of distributed solutions over the past decade established a componentized application design model that, along with the Java programming language, natively supports object-orientation. As a result, reusability is achievable on a component level. Because service-orientation encourages services to be reusable and because a service can encapsulate one or more new or existing EJB components, reusability on a service level comes down to the design of a service's business logic and endpoint.

**Statelessness.** JAX-RPC Service Endpoints can be designed to exist as stateless servlets, but the JAX-RPC API does provide the means for the servlet to manage state information through the use of the HttpSession object. It is therefore up to the service designer to ensure that statelessness is maximized and session information is only persisted in this manner when absolutely necessary. As previously mentioned, one of the requirements for adapting an EJB component into an EJB Service Endpoint is that it be completely stateless. In the J2EE world, this means that it must be designed as a Stateless Session Bean, a type of EJB that does not manage state but that

may still defer state management to other types of EJB components (such as Stateful Session Beans or Entity Beans).

**Discoverability.** As with reuse, service discoverability requires deliberate design. To make a service discoverable, the emphasis is on the endpoint design, in that it must be as descriptive as possible. Service discovery as part of a J2EE SOA is directly supported through JAXR, an API that provides a programmatic interface to XML-based registries, including UDDI repositories. The JAXR library consists of two separate APIs for publishing and issuing searches against registries. Note that even if JAXR is used to represent a UDDI registry, it does so by exposing an interface that differs from the standard UDDI API. (For example, a UDDI Business-Entity is a JAXR Organization, and a UDDI BusinessService is a JAXR Service.)

**Interoperability** is, to a large extent, a quality deliberately designed into a Web service. Aside from service interface design characteristics, conformance to industry-standard Web services specifications is critical to achieving interoperable SOAs, especially when interoperability is required across enterprise domains. As of version 1.1, the JAX-RPC API is fully capable of creating WS-I Basic Profile-compliant Web services. This furthers the vision of producing services that are intrinsically interoperable. Care must be taken, though, to prevent the use of handlers from performing runtime processing actions that could jeopardize this compliance. IBM's Rational Application Developer provides built-in support for building WS-I compliant Web services. Further, the Wscompile tool, which is part of the J2EE SDK from Sun Microsystems, allows for the auto-generation of WS-I Basic Profile compliant WSDL definitions.

**Federation.** Strategically positioned services coupled with adapters that expose legacy application logic can establish a degree of federation. Building an integration architecture with custom business services and legacy wrapper services can be achieved using basic J2EE APIs and features. Supplementing such an architecture with an orchestration server (and an accompanying orchestration service layer) further increases the potential of unifying and standardizing integrated logic. Also worth taking into consideration is the J2EE Connector Architecture (JCA), a structured, adapter-centric integration architecture through which resource adapters are used to bridge gaps between J2EE platforms and other environments. As with JMS, JCA is traditionally centered around the use of proprietary messaging protocols and platform-specific adapters. Recently, however, support for asynchronous and SOAP messaging has been introduced. Further, service adapters have been made available to tie JCA environments into service-oriented solutions. Numerous integration server platforms also are available to support and implement the overall concept of enterprise-wide federation. Depending on the nature of the integration architecture, service-oriented integration environments are built around orchestration servers or enterprise service bus offerings (or both). The Sun ONE Connector Builder product is an example of a vendor implementation of JCA that supports the creation of a SOAP messaging layer. Also a number of J2EE vendors provide orchestration servers with native WS-BPEL support, including IBM's WebSphere Business Integration Server Foundation product and Oracle's BPEL Process Manager.

**Composable.** Given the modular nature of supporting API packages and classes and the choice of service-specific containers, the J2EE platform is intrinsically composable. This allows solution designers to use only the parts of the platform required for a particular application. For example, a Web services solution that only consists of JAX-RPC Service Endpoints will likely not have a need for the JMS class packages or a J2EE SOA that does not require a service registry will not implement any part of the JAXR API. With regards to taking advantage of the composable contemporary SOA landscape, the J2EE platform, in its current incarnation, does not yet provide native support for WS-\* specifications. Instead, extensions are supplied by product vendors that implement and build upon J2EE standards. The extent to which the WS-\* features of an SOA based on the J2EE platform can be composed is therefore currently dependent upon the vendor-specific platform used.

**Externsibility.** As with any service-oriented solution, those based on the J2EE platform can be designed with services that support the notion of future extensibility. This comes down to fundamental design characteristics that impose conventions and structure on the service interface level. Because J2EE environments are implemented by different vendors, extensibility can sometimes lead to the use of proprietary extensions. While still achieving extensibility within the vendor environment, this can limit the portability and openness of Java solutions.

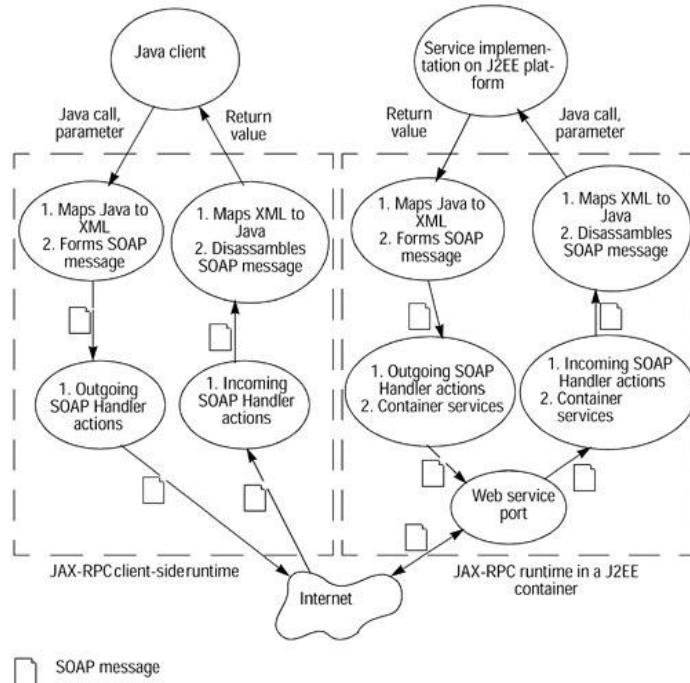
**Service-oriented business modeling.** Beyond consistent and standardized design approaches to building service layers along the lines of the application, entity, and task-centric services we've established in previous chapters, there is no inherent support for service-oriented business modeling within J2EE. This is primarily because the concept of orchestration is not a native part of the J2EE platform. Instead, orchestration services and design tools are provided by vendors to supplement the J2EE Web services development and runtime environment. Service-oriented business modeling and the service layers we've discussed in this book can therefore be created with the right vendor tools.

**Logic-level abstraction.** JAX-RPC Service Endpoints and EJB Service Endpoints can be designed into service layers that abstract application-specific or reusable logic. Further, entire J2EE solutions can be exposed through these types of services, when appropriate. Depending on the vendor server platform used, some limitations may be encountered when building service compositions that require message-level security measures. These limitations may inhibit the extent of feasible logic-level abstraction.

### 3.3.10.8 Flow of a Web Service Call

In a Web service scenario, a client makes a request to a particular Web service, such as asking for the weather at a certain location, and the service, after processing the request, sends a response to the client to fulfill the request. When both the client and the Web service are implemented in a Java environment, the client makes the call to the service by invoking a Java method, along with setting up and passing the required parameters, and receives as the response the result of the method invocation.

To help you understand the context within which you design Web services, let's first take a high-level view at what happens beneath the hood in a typical Web services implementation in a Java environment. Figure 3.36 shows how a Java client communicates with a Java Web service on the J2EE platform.



**Figure 3.36 Flow of a Web Service on Java Platform**

Figure 3.36 changes when a non-Java client interacts with a Java Web service. In such a case, the right side of the figure, which reflects the actions of the Web service, stays the same as depicted here, but the left side of the figure would reflect the actions of the client platform. When a Java client invokes a Web service that is on a non-Java platform, the right side of the figure changes to reflect the Web service platform and the left side, which reflects the actions of the client, remains as shown in the figure.

Once the client knows how to access the service, the client makes a request to the service by invoking a Java method, which is passed with its parameters to the client-side JAX-RPC runtime. With the method call, the client is actually invoking an operation on the service. These operations represent the different services of interest to clients. The JAX-RPC runtime maps the Java types to standard XML types and forms a SOAP message that encapsulates the method call and parameters. The runtime then passes the SOAP message through the SOAP handlers, if there are any, and then to the server-side service port.

The client's request reaches the service through a port, since a port provides access over a specific protocol and data format at a network endpoint consisting of a host name and port number.

Before the port passes the request to the endpoint, it ensures that the J2EE container applies its declarative services (such as security checks) to the SOAP request. After that, any developer-written SOAP handlers in place are applied to the request. Note that SOAP handlers, which are optional, let developers apply application-specific processing logic common to all requests and responses that flow through this endpoint. After the handlers operate on the SOAP message, the message is passed to the service endpoint.

The J2EE container extracts the method call invoked by the client along with the parameters for the call, performs any XML-to-Java object mapping necessary, and hands the method to the Web service interface implementation for further processing. A similar set of steps happens when the service sends back its response.

### 3.3.10.9 Designing the Interface

There are some considerations to keep in mind as you design the interface of your Web service, such as issues regarding overloading methods, choosing the endpoint type, and so forth. Before examining these issues, decide on the approach you want to take for developing the service's interface definition.

Two approaches to developing the interface definition for a Web service are:

1. Java-to-WSDL— Start with a set of Java interfaces for the Web service and from these create the Web Services Description Language (WSDL) description of the service for others to use.
2. WSDL-to-Java— Start with a WSDL document describing the details of the Web service interface and use this information to build the corresponding Java interfaces.

How do these two approaches compare? Starting with Java interfaces and creating a WSDL document is probably the easier of the two approaches. With this approach, you need not know any WSDL details because you use vendor-provided tools to create the WSDL description. While these tools make it easy for you to generate WSDL files from Java interfaces, you do lose some control over the WSDL file creation.

With the Java-to-WSDL approach, keep in mind that the exposed service interface may be too unstable from a service evolution point of view. With the Java-to-WSDL approach, it may be hard to evolve the service interface without forcing a change in the corresponding WSDL document, and changing the WSDL might require rewriting the service's clients. These changes, and the accompanying instability, can affect the interoperability of the service itself. Since achieving interoperability is a prime reason to use Web services, the instability of the Java-to-WSDL approach is a major drawback. Also, keep in mind that different tools may use different interpretations for certain Java types (for example, `java.util.Date` might be interpreted as `java.util.Calendar`), resulting in different representations in the WSDL file. While not common, these representation variations may result in some semantic surprises.

On the other hand, the WSDL-to-Java approach gives you a powerful way to expose a stable service interface that you can evolve with relative ease. Not only does it give you greater design flexibility, the WSDL-to-Java approach also provides an ideal way for you to finalize all service details—from method call types and fault types to the schemas representing exchanged business documents—before you even start a service or client implementation.

**Choice of the Interface Endpoint Type.** In the J2EE platform, you have two choices for implementing the Web service interface—you can use a JAX-RPC service endpoint (also referred to as a Web tier endpoint) or an EJB service endpoint (also referred to as an EJB tier endpoint). Using one of these endpoint types makes it possible to embed the endpoint in the same tier as the service implementation. This simplifies the service implementation, because it obviates the need to place the endpoint in its own tier where the presence of the endpoint is solely to act as a proxy directing requests to other tiers that contain the service's business logic.

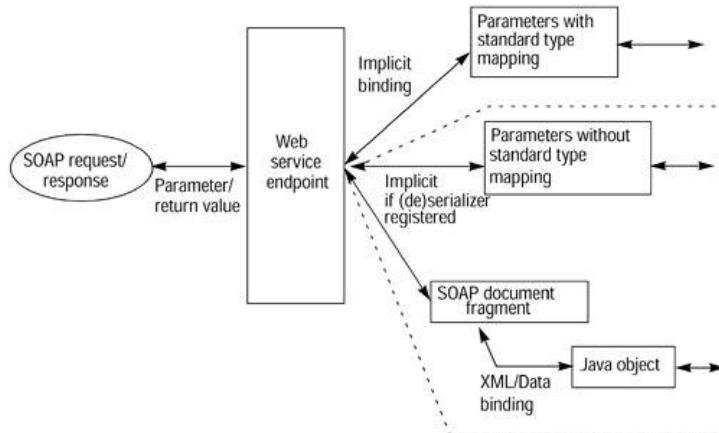
When you develop a new Web service that does not use existing business logic, choosing the endpoint type to use for the Web service interface is straightforward. The endpoint type choice depends on the nature of your business logic—whether the business logic of the service is completely contained within either the Web tier or the EJB tier:

- Use a JAX-RPC service endpoint when the processing layer is within the Web tier.
- Use an EJB service endpoint when the processing layer is only on the EJB tier.

**Parameter Types for Web Service Operations.** A Web service interface exposes a set of method calls to clients. When invoking a service interface method, a client may have to set values for the parameters associated with the

call. When you design an interface's methods, choose carefully the types of these parameters. A method call and its parameters are sent as a SOAP message between the client and the service. To be part of a SOAP message, parameters must be mapped to XML. When received at the client or service end, the same parameters must be mapped from XML to their proper types or objects. This section describes some guidelines to keep in mind when defining method call parameters and return values.

Parameters for Web service method calls may be standard Java objects and types, XML documents, or even nonstandard types. Whether you use the Java-to-WSDL approach or the WSDL-to-Java approach, each type of parameter must be mapped to its XML equivalent in the SOAP message. Figure 3.37 shows how the binding happens for various types of parameters.



**Figure 3.37. Binding Parameters and Return Values with JAX-RPC**

**Java Objects as Parameters.** Parameters for Web service calls can be standard Java types and objects. If you use the Java-to-WSDL approach, you specify the parameter types as part of the arguments of the method calls of your Java interface. If you use the WSDL-to-Java approach, you specify the parameter types as the type or element attributes of the part element of each message in your WSDL. The type of a parameter that you use has a significant effect on the portability and interoperability of your service.

The platform supports the following Java data types:

- Java primitive types boolean, byte, short, int, long, float, and double, along with their corresponding wrapper Java classes
- Standard Java classes: String, Date, Calendar, BigInteger, BigDecimal, QName, and URI
- Java arrays with JAX-RPC-supported Java types as members
- JAX-RPC value types—user-defined Java classes, including classes with JavaBeans component-like properties

When designing parameters for method calls in a Web service interface, choose parameters that have standard type mappings. (See Figure 3.37) Always keep in mind that the portability and interoperability of your service is reduced when you use parameter types that by default are not supported. As Figure 3.37 shows, parameters that have standard type mappings are bound implicitly. However, the developer must do more work when using parameters that do not have standard type mappings.

Since the J2EE container automatically handles mappings based on the Java types, using these Java-MIME mappings frees you from the intricacies of sending and retrieving documents and images as part of a service's request and response handling. For example, your service, if it expects to receive a GIF image with a MIME type of image/gif, can expect the client to send a java.awt.Image object. A sample Web service interface that receives an image might look like the one shown in Code Example 3.39:

#### **Example 3.39. Receiving a java.awt.Image Object**

```

import java.awt.Image;
public interface WeatherMapService extends Remote {
    public void submitWeatherMap(Image weatherMap)
        throws RemoteException, InvalidMapException; }

```

In this example, the Image object lets the container implementation handle the image-passing details. The container provides javax.activation.DataHandler classes, which work with the Java Activation Framework to accomplish the Java-MIME and MIME-Java mappings.

Considering this mapping between Java and MIME types, it is best to send images and XML documents that are in a Web service interface using the Java types. However, you should be careful about the effect on the interoperability of your service.

**XML Documents as Parameters.** There are scenarios when you want to pass XML documents as parameters. Typically, these occur in business-to-business interactions where there is a need to exchange legally binding business documents, track what is exchanged, and so forth. Exchanging XML documents as part of a Web service is addressed in a separate section.

**Handling Nonstandard Type Parameters.** JAX-RPC technology, in addition to providing a rich standard mapping set between XML and Java data types, also provides an extensible type mapping framework. Developers can use this framework to specify pluggable, custom serializers and deserializers that support nonstandard type mappings. Extensible type mapping frameworks, which developers may use to support nonstandard type mappings, are not yet a standard part of the J2EE platform.

Vendors currently can provide their own solutions to this problem. It must be emphasized that if you implement a service using some vendor's implementation-specific type mapping framework, then your service is not guaranteed to be portable and interoperable.

Because of portability limitations, you should avoid passing parameters that require the use of vendor-specific serializers or deserializers. Instead, a better way is to pass these parameters as SOAP document fragments represented as a DOM subtree in the service endpoint interface. (See Figure 3.37) If so, you should consider binding (either manually or using JAXB) the SOAP fragments to Java objects before passing them to the processing layer to avoid tightly coupling the business logic with the document fragment.

**Interfaces with Overloaded Methods.** In your service interface, you may overload methods and expose them to the service's clients. Overloaded methods share the same method name but have different parameters and return values. If you do choose to use overloaded methods as part of your service interface, keep in mind that there are some limitations, as follows:

- If you choose the WSDL-to-Java approach, there are limitations to representing overloaded methods in a WSDL description. In the WSDL description, each method call and its response are represented as unique SOAP messages. To represent overloaded methods, the WSDL description would have to support multiple SOAP messages with the same name. WSDL version 1.1 does not have this capability to support multiple messages with the same name.
- If you choose the Java-to-WSDL approach and your service exposes overloaded methods, be sure to check how any vendor-specific tools you are using represent these overloaded methods in the WSDL description. You need to ensure that the WSDL representation of overloaded methods works in the context of your application.

Let's see how this applies in the weather service scenario. As the provider, you might offer the service to clients, letting them look up weather information by city name or zip code. If you use the Java-to-WSDL approach, you might first define the WeatherService interface as shown in Code Example 3.40.

#### ***Example 3.40. WeatherService Interface for Java-to-WSDL Approach***

```
public interface WeatherService extends Remote {  
    public String getWeather(String city) throws RemoteException;  
    public String getWeather(int zip) throws RemoteException;  
}
```

After you define the interface, you run the vendor-provided tool to create the WSDL from the interface. Each tool has its own way of representing the getWeather overloaded methods in the WSDL, and your WSDL reflects the particular tool you use. For example, if you use the J2EE 1.4 SDK from Sun Microsystems, its wscompile tool creates from the WeatherService interface the WSDL shown in Code Example 3.41.

#### **Example 3.41. Generated WSDL for WeatherService Interface**

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WeatherWebService" ....>
    <types/>
    <message name="WeatherService_getWeather">
        <part name="int_1" type="xsd:int"/>
    </message>
    <message name="WeatherService_getWeatherResponse">
        <part name="result" type="xsd:string"/>
    </message>
    <message name="WeatherService_getWeather2">
        <part name="String_1" type="xsd:string"/>
    </message>
    <message name="WeatherService_getWeather2Response">
        <part name="result" type="xsd:string"/>
    </message>
    ...
</definitions>
```

Notice that the WSDL represents the getWeather overloaded methods as two different SOAP messages, naming one getWeather, which takes an integer for the zip code as its parameter, and the other getWeather2, which takes a string parameter for the city. As a result, a client interested in obtaining weather information using a city name invokes the service by calling getWeather2, as shown in Code Example 3.42.

#### **Example 3.42. Using Weather Service Interface with Java-to-WSDL Approach**

```
Context ic = new InitialContext();
WeatherWebService weatherSvc = (WeatherWebService)
    ic.lookup("java:comp/env/service/WeatherService");
WeatherServiceIntf port = (WeatherServiceIntf)
    weatherSvc.getPort(WeatherServiceIntf.class);
String returnValue = port.getWeather2("San Francisco");
```

For example, to obtain the weather information for San Francisco, the client called port.getWeather2("Timisoara"). Keep in mind that another tool may very likely generate a WSDL whose representation of overloaded methods is different.

You may want to avoid using overloaded methods in your Java interface altogether if you prefer to have only intuitive method names in the WSDL. If instead you choose to use the WSDL-to-Java approach, your WSDL description might look as follows. (See Code Example 3.43)

#### **Example 3.43. WSDL for Weather Service with Overloaded Methods Avoided**

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions name="WeatherWebService" ...>
    <types/>
    <message name="WeatherService_getWeatherByZip">
        <part name="int_1" type="xsd:int"/>
    </message>
    <message name="WeatherService_getWeatherByZipResponse">
        <part name="result" type="xsd:string"/>
    </message>
    <message name="WeatherService_getWeatherByCity">
        <part name="String_1" type="xsd:string"/>
    </message>
    <message name="WeatherService_getWeatherByCityResponse">
        <part name="result" type="xsd:string"/>
    </message>
    ...
</definitions>
```

Since the messages in a WSDL file must have unique names, you must use different message names to represent methods that you would otherwise overload. These different message names actually convert to different method calls in your interface. Notice that the WSDL includes a method `getWeatherByZip`, which takes an integer parameter, and a method `getWeatherByCity`, which takes a string parameter. Thus, a client wishing to obtain weather information by city name from a `WeatherService` interface associated with the WSDL in Code Example 3.43 might invoke the service as shown in Code Example 3.44.

#### **Example 3.44. Using Weather Service with WSDL-to-Java Approach**

```
Context ic = new InitialContext();
WeatherWebService weatherSvc = (WeatherWebService)
    ic.lookup("java:comp/env/service/WeatherService");
WeatherServiceIntf port = (WeatherServiceIntf)
    weatherSvc.getPort(WeatherServiceIntf.class);
String returnValue = port.getWeatherByCity("Timisoara");
```

**Handling Exceptions.** Just like any Java or J2EE application, a Web service application may encounter an error condition while processing a client request. A Web service application needs to properly catch any exceptions thrown by an error condition and propagate these exceptions. For a Java application running in a single virtual machine, you can propagate exceptions up the call stack until reaching a method with an exception handler that handles the type of exception thrown. To put it another way, for non-Web service J2EE and Java applications, you may continue to throw exceptions up the call stack, passing along the entire stack trace, until reaching a method with an exception handler that handles the type of exception thrown. You can also write exceptions that extend or inherit other exceptions.

However, throwing exceptions in Web service applications has additional constraints that impact the design of the service endpoint. When considering how the service endpoint handles error conditions and notifies clients of errors, you must keep in mind these points:

- Similar to requests and responses, exceptions are also sent back to the client as part of the SOAP messages.
- Your Web service application should support clients running on non-Java platforms that may not have the same, or even similar, error-handling mechanisms as the Java exception-handling mechanism.

A Web service application may encounter two types of error conditions. One type of error might be an irrecoverable system error, such as an error due to a network connection problem. When an error such as this occurs, the JAX-RPC runtime on the client throws the client platform's equivalent of an irrecoverable system exception. For Java clients, this translates to a `RemoteException`.

A Web service application may also encounter a recoverable application error condition. This type of error is called a service-specific exception. The error is particular to the specific service. For example, a weather Web service might indicate an error if it cannot find weather information for a specified city.

To illustrate the Web service exception-handling mechanism, let's examine it in the context of the weather Web service example. When designing the weather service, you want the service to be able to handle a scenario in which the client requests weather information for a nonexistent city. You might design the service to throw a service-specific exception, such as `CityNotFoundException`, to the client that made the request. You might code the service interface so that the `getWeather` method throws this exception. (See Code Example 3.45)

#### **Example 3.45. Throwing a Service-Specific Exception**

```
public interface WeatherService extends Remote {
    public String getWeather(String city) throws
        CityNotFoundException, RemoteException;}
```

Service-specific exceptions like `CityNotFoundException`, which are thrown by the Web service to indicate application-specific error conditions, must be checked exceptions that directly or indirectly extend `java.lang.Exception`. They cannot be unchecked exceptions. Code Example 3.46 shows a typical implementation of a service-specific exception, such as for `CityNotFoundException`.

#### **Example 3.46. Implementation of a Service-Specific Exception**

```
public class CityNotFoundException extends Exception {  
    private String message;  
    public CityNotFoundException(String message) {  
        super(message);  
        this.message = message;  
    }  
    public String getMessage() {  
        return message;  
    }  
}
```

Code Example 3.47 shows the service implementation for the same weather service interface. This example illustrates how the service might throw CityNotFoundException.

#### **Example 3.47. Example of a Service Throwing a Service-Specific Exception**

```
public class WeatherServiceImpl implements WeatherService {  
    public String getWeather(String city)  
        throws CityNotFoundException {  
        if(!validCity(city))  
            throw new CityNotFoundException(city + " not found");  
        // Get weather info and return it back  
    }  
}
```

Convert application-specific errors and other Java exceptions into meaningful service-specific exceptions and throw these service-specific exceptions to the clients.

Although they promote interoperability among heterogeneous platforms, Web service standards cannot address every type of exception thrown by different platforms. For example, the standards do not specify how Java exceptions such as java.io.IOException and javax.ejb.EJBException should be returned to the client. As a consequence, it is important for a Web service—from the service's interoperability point of view—to not expose Java-specific exceptions (such as those just mentioned) in the Web service interface. Instead, throw a service-specific exception. In addition, keep the following points in mind:

- You cannot throw nonserializable exceptions to a client through the Web service endpoint.
- When a service throws java or javax exceptions, the exception type and its context information are lost to the client that receives the thrown exception. For example, if your service throws a javax.ejb.FinderException exception to the client, the client may receive an exception named FinderException, but its type information may not be available to the client. Furthermore, the type of the exception to the client may not be the same as the type of the thrown exception. (Depending on the tool used to generate the client-side interfaces, the exception may even belong to some package other than javax.ejb.)

As a result, you should avoid directly throwing java and javax exceptions to clients. Instead, when your service encounters one of these types of exceptions, wrap it within a meaningful service-specific exception and throw this service-specific exception back to the client. For example, suppose your service encounters a javax.ejb.FinderException exception while processing a client request. The service should catch the FinderException exception, and then, rather than throwing this exception as is back to the client, the service should instead throw a service-specific exception that has more meaning for the client. See Code Example 3.48.

#### **Example 3.48. Converting an Exception into a Service-Specific Exception**

```
try {  
    // findByPrimaryKey  
    // Do processing  
    // return results  
} catch (javax.ejb.FinderException fe) {
```

```

        throw new InvalidKeyException(
            "Unable to find row with given primary key");
    }
}

```

Exception inheritances are lost when you throw a service-specific exception.

You should avoid defining service-specific exceptions that inherit or extend other exceptions. For example, if CityNotFoundException in Code Example DES.E22 extends another exception, such as RootException, then when the service throws CityNotFoundException, methods and properties inherited from RootException are not passed to the client. The exception stack trace is not passed to the client.

The stack trace for an exception is relevant only to the current execution environment and is meaningless on a different system. Hence, when a service throws an exception to the client, the client does not have the stack trace explaining the conditions under which the exception occurred. Thus, you should consider passing additional information in the message for the exception.

Web service standards make it easier for a service to pass error conditions to a client in a platform-independent way. While the following discussion may be of interest, it is not essential that developers know these details about the J2EE platform's error-handling mechanisms for Web services.

As noted previously, error conditions are included within the SOAP messages that a service returns to clients. The SOAP specification defines a message type, called fault, that enables error conditions to be passed as part of the SOAP message yet still be differentiated from the request or response portion. Similarly, the WSDL specification defines a set of operations that are possible on an endpoint. These operations include input and output operations, which represent the request and response respectively, and an operation called fault.

A SOAP fault defines system-level exceptions, such as RemoteException, which are irrecoverable errors. The WSDL fault denotes service-specific exceptions, such as CityNotFoundException, and these are recoverable application error conditions. Since the WSDL fault denotes a recoverable error condition, the platform can pass it as part of the SOAP response message. Thus, the standards provide a way to exchange fault messages and map these messages to operations on the endpoint.

Code Example 3.49 shows the WSDL code for the same weather Web service example. This example illustrates how service-specific exceptions are mapped just like input and output messages are mapped.

#### ***Example 3.49. Mapping a Service-Specific Exception in WSDL***

```

<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
    ...
    <message name="WeatherService_getWeather">
        <part name="String_1" type="xsd:string"/>
    </message>
    <message name="WeatherService_getWeatherResponse">
        <part name="result" type="xsd:string"/>
    </message>
    <message name="CityNotFoundException">
        <part name="CityNotFoundException"
              element="tns:CityNotFoundException"/>
    </message>
    <portType name="WeatherService">
        <operation name="getWeather" parameterOrder="String_1">
            <input message="tns:WeatherService_getWeather"/>
            <output message=
                    "tns:WeatherService_getWeatherResponse"/>
            <fault name="CityNotFoundException"
                  message="tns:CityNotFoundException"/>
        </operation>
    </portType>
    ...
</definitions>

```

**Use of Handlers.** JAX-RPC technology enables you to plug in SOAP message handlers, thus allowing processing of SOAP messages that represent requests and responses. Plugging in SOAP message handlers gives you the capability to examine and modify the SOAP requests before they are processed by the Web service and to examine and modify the SOAP responses before they are delivered to the client.

Handlers are particular to a Web service and are associated with the specific port of the service. As a result of this association, the handler's logic applies to all SOAP requests and responses that pass through a service's port. Thus, you use these message handlers when your Web service must perform some SOAP message-specific processing common to all its requests and responses. Because handlers are common to all requests and responses that pass through a Web service endpoint, keep the following guideline in mind:

It is not advisable to put in a handler business logic or processing particular to specific requests and responses. You cannot store client-specific state in a handler: A handler's logic acts on all requests and responses that pass through an endpoint. However, you may use the handler to store port-specific state, which is state common to all method calls on that service interface. Note also that handlers execute in the context of the component in which they are present.

Do not store client-specific state in a handler. Also note that handlers work directly on the SOAP message, and this involves XML processing. You can use handlers to pass client-specific state through the message context.

Use of handlers can result in a significant performance impact for the service as a whole. Use of handlers could potentially affect the interoperability of your service. See the next section on interoperability. Keep in mind that it takes advanced knowledge of SOAP message manipulation APIs (such as SAAJ) to correctly use handlers. To avoid errors, Web service developers should try to use existing or vendor-supplied handlers. Using handlers makes sense primarily for writing system services such as auditing, logging, and so forth.

**Interoperability.** A major benefit of Web services is interoperability between heterogeneous platforms. To get the maximum benefit, you want to design your Web service to be interoperable with clients on any platform, and the Web Services Interoperability (WS-I) organization helps in this regard. WS-I promotes a set of generic protocols for the interoperable exchange of messages between Web services. The WS-I Basic Profile promotes interoperability by defining and recommending how a set of core Web services specifications and standards (including SOAP, WSDL, UDDI, and XML) can be used for developing interoperable Web services.

In addition to the WS-I protocols, other groups, such as SOAPBuilders Interoperability group (see <http://java.sun.com/wsinterop/sb/index.html>), provide common testing grounds that make it easier to test the interoperability of various SOAP implementations. This has made it possible for various Web services technology vendors to test the interoperability of implementations of their standards. When you implement your service using technologies that adhere to the WS-I Basic Profile specifications, you are assured that such services are interoperable.

Apart from these standards and testing environments, you as the service developer must design and implement your Web service so that maximum interoperability is possible. For maximum interoperability, you should keep these three points in mind:

1. The two messaging styles and bindings supported by WSDL
2. The WS-I support for attachments
3. The most effective way to use handlers

WSDL supports two types of messaging styles: rpc and document. The WSDL style attribute indicates the messaging style. (See Code Example 3.50) A style attribute set to rpc indicates a RPC-oriented operation, where messages contain parameters and return values, or function signatures. When the style attribute is set to document, it indicates a document-oriented operation, one in which messages contain documents. Each operation style has a different effect on the format of the body of a SOAP message.

#### **Example 3.50. Specifying WSDL Bindings**

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions .....>
<binding name="WeatherServiceBinding" type="tns:WeatherService">
    <operation name="getWeather">
```

```

<input>
    <soap:body use="literal"
        namespace="urn:WeatherWebService"/>
</input>
<output>
    <soap:body use="literal"
        namespace="urn:WeatherWebService"/>
</output>
<soap:operation soapAction="" /></operation>
<soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http" />
</binding>
<service . . . . .>
</definitions>

```

Along with operation styles, WSDL supports two types of serialization and deserialization mechanisms: a literal and an encoded mechanism. The WSDL use attribute indicates which mechanism is supported. (See Code Example 3.50. A literal value for the use attribute indicates that the data is formatted according to the abstract definitions within the WSDL document. The encoded value means data is formatted according to the encodings defined in the URI specified by the encodingStyle attribute. Thus, you can choose between an rpc or document style of message passing and each message can use either a literal or encoded data formatting.

Because the WS-I Basic Profile to which J2EE platform conforms, supports only literal bindings, you should avoid encoded bindings. Literal bindings cannot represent complex types, such as objects with circular references, in a standard way.

Code Example DES.E26 shows a snippet from the WSDL document illustrating how the sample weather service specifies these bindings.

It is important to keep in mind these message styles and bindings, particularly when you design the interface using the WSDL-to-Java approach and when you design the WSDL for your service. When you use the Java-to-WSDL approach, you rely on the vendor-provided tools to generate the WSDL for your Java interfaces, and they can be counted on to create WS-I-compliant WSDL for your service. However, note that some vendors may expect you to specify certain options to ensure the creation of a WS-I-compliant WSDL. For example, the J2EE 1.4 SDK from Sun Microsystems provides a wscompile tool, which expects the developer to use the -f:wsi flag to create the WS-I-compliant WSDL for the service. It is also a good idea to check the WSDL document itself to ensure that whatever tool you use created the document correctly.

### 3.3.10.10 Receiving Requests

The interaction layer, through the endpoint, receives client requests. The platform maps the incoming client requests, which are in the form of SOAP messages, to method calls present in the Web service interface.

Web service calls are basically method calls whose parameters are passed as either Java objects, XML documents (`javax.xml.transform.Source` objects), or even SOAP document fragments (`javax.xml.soap.SOAPElement` objects).

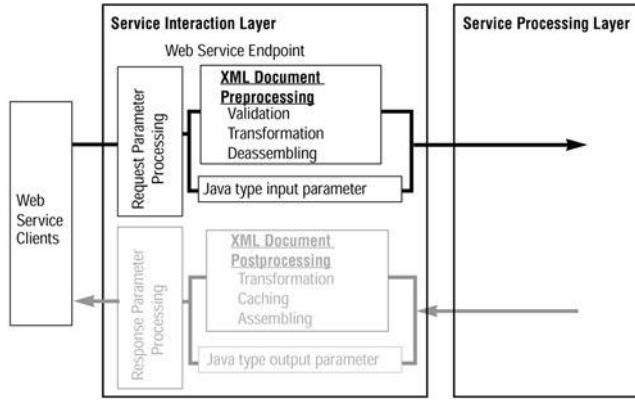
For parameters that are passed as Java objects (such as `String`, `int`, JAX-RPC value types, and so forth), do the application-specific parameter validation and map the incoming objects to domain-specific objects in the interaction layer before delegating the request to the processing layer.

You may have to undertake additional steps to handle XML documents that are passed as parameters. These steps, which are best performed in the interaction layer of your service, are as follows:

1. The service endpoint should validate the incoming XML document against its schema.
2. When the service's processing layer and business logic are designed to deal with XML documents, you should transform the XML document to an internally supported schema, if the schema for the XML document differs from the internal schema, before passing the document to the processing layer.
3. When the processing layer deals with objects but the service interface receives XML documents, then, as part of the interaction layer, map the incoming XML documents to domain objects before delegating the request to the processing layer.

It is important that these three steps—validation of incoming parameters or XML documents, translation of XML documents to internal supported schemas, and mapping documents to domain objects—be performed as close to the service endpoint as possible, and certainly in the service interaction layer.

A design such as this helps to catch errors early, and thus avoids unnecessary calls and round-trips to the processing layer. Figure 3.38 shows the recommended way to handle requests and responses in the Web service's interaction layer.



**Fig. 3.38. Web Service Request Processing**

The Web service's interaction layer handles all incoming requests and delegates them to the business logic exposed in the processing layer. When implemented in this manner, the Web service interaction layer has several advantages, since it gives you a common location for the following tasks:

- Managing the handling of requests so that the service endpoint serves as the initial point of contact
- Invoking security services, including authentication and authorization
- Validating and transforming incoming XML documents and mapping XML documents to domain objects
- Delegating to existing business logic
- Handling errors

It is generally advisable to do all common processing—such as security checks, logging, auditing, input validation, and so forth—for requests at the interaction layer as soon as a request is received and before passing it to the processing layer.

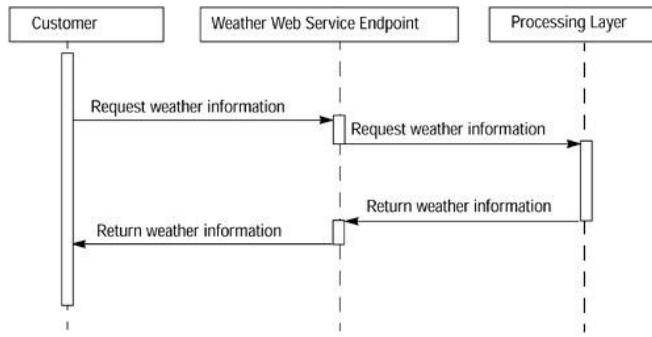
**Delegating Web Services Requests to Processing Layer.** After designing the request preprocessing tasks, the next step is to design how to delegate the request to the processing layer. At this point, consider the kind of processing the request requires, since this helps you decide how to delegate the request to the processing layer. All requests can be categorized into two large categories based on the time it takes to process the request, namely:

- A request that is processed in a short enough time so that a client can afford to block and wait to receive the response before proceeding further. In other words, the client and the service interact in a synchronous manner such that the invoking client blocks until the request is processed completely and the response is received.
- A request that takes a long time to be processed, so much so that it is not a good idea to make the client wait until the processing is completed. In other words, the client and the service interact in an asynchronous manner such that the invoking client need not block and wait until the request is processed completely.

When referring to request processing, we use the terms synchronous and asynchronous from the point of view of when the client's request processing completes fully. Keep in mind that, under the hood, an asynchronous interaction between a client and a service might result in a synchronous invocation over the network, since HTTP is by its nature synchronous. Similarly, SOAP messages sent over HTTP are also synchronous.

The weather information service is a good example of a synchronous interaction between a client and a service. When it receives a client's request, the weather service must look up the required information and send back a response to the client. This look-up and return of the information can be achieved in a relatively short time, during which the client can be expected to block and wait. The client continues its processing only after it obtains a response from the service. (See Figure 3.39.)

A Web service such as this can be designed using a service endpoint that receives the client's request and then delegates the request directly to the service's appropriate logic in the processing layer. The service's processing layer processes the request and, when the processing completes, the service endpoint returns the response to the client.



**Fig. 3.39. Weather Information Service Interaction**

Code Example 3.51 shows the weather service interface performing some basic parameter validation checks in the interaction layer. The interface also gets required information and passes that information to the client in a synchronous manner:

#### **Example 3.51. Performing a Synchronous Client Interaction**

```

public class WeatherServiceImpl implements WeatherService, ServiceLifecycle {
    public void init(Object context) throws JAXRPCException {....}
    public String getWeather(String city) throws CityNotFoundException {
        /** Validate parameters */
        if(!validCity(city)) throw new CityNotFoundException(....);
        /** Get weather info from processing layer and return results */
        return (getWeatherInfoFromDataSource(city));
    }
    public void destroy() {....}
}
  
```

Now let's examine an asynchronous interaction between a client and a service. When making a request for this type of service, the client cannot afford to wait for the response because of the significant time it takes for the service to process the request completely. Instead, the client may want to continue with some other processing. Later, when it receives the response, the client resumes whatever processing initiated the service request. Typically in these types of services, the content of the request parameters initiates and determines the processing workflow—the steps to fulfill the request—for the Web service. Often, fulfilling a request requires multiple workflow steps.

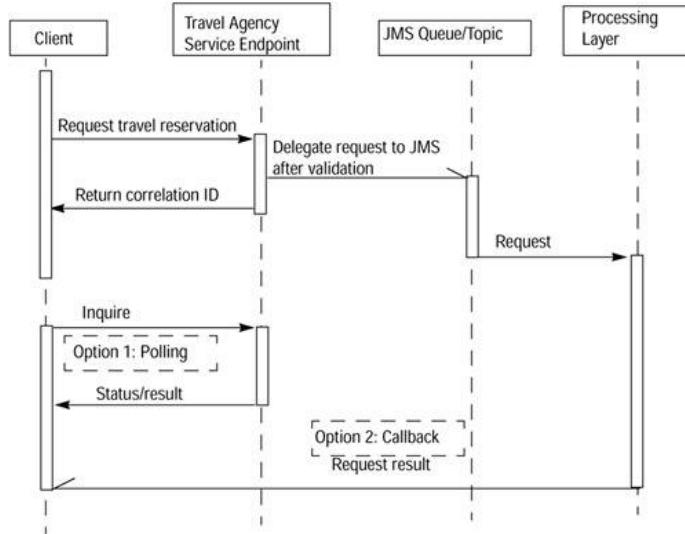
The travel agency service is a good example of an asynchronous interaction between a client and a service. A client requests arrangements for a particular trip by sending the travel service all pertinent information (most likely in an XML document). Based on the document's content, the service performs such steps as verifying the user's account, checking and getting authorization for the credit card, checking accommodations and transportation availability, building an itinerary, purchasing tickets, and so forth. Since the travel service must perform a series of often time-consuming steps in its normal workflow, the client cannot afford to pause and wait for these steps to complete.

One recommended approach for asynchronously delegating these types of Web service requests to the processing layer in the following: In this architecture, the client sends a request to the service endpoint. The service endpoint validates the incoming request in the interaction layer and then delegates the client's request to the appropriate processing layer of the service. It does so by sending the request as a JMS message to a JMS queue or topic specifically designated for this type of request.

Delegating a request to the processing layer through JMS before validating the request should be avoided. Validation ensures that a request is correct. Delegating the request before validation may result in passing an invalid request to the processing layer, making error tracking and error handling overly complex. After the request

is successfully delegated to the processing layer, the service endpoint may return a correlation identifier to the client. This correlation identifier is for the client's future reference and may help the client associate a response that corresponds to its previous request. If the business logic is implemented using enterprise beans, message-driven beans in the EJB tier read the request and initiate processing so that a response can ultimately be formulated.

Figure 3.40 shows how the travel agency service might implement this interaction, and Code Example 3.52 shows the actual code that might be used.



*Fig. 3.40. Travel Agency Service Interaction*

#### **Example 3.52. Implementing Travel Agency Service Interaction**

```

public class ReservationRequestRcvr {
    public ReservationRequestRcvr() throws RemoteException {....}
    public String receiveRequest(Source reservationDetails) throws
        RemoteException, InvalidRequestException{
        /** Validate incoming XML document **/
        String xmlDoc = getDocumentAsString(reservationDetails);
        if(!validDocument(xmlDoc))
            throw new InvalidRequestException(...);
        /** Get a JMS Queue and delegate the incoming request to the queue ***/
        QueueConnectionFactory queueFactory =
            serviceLocator.getQueueConnectionFactory(...);
        Queue reservationRequestQueue = serviceLocator.getQueue(...);
        QueueConnection connection = queueFactory.createQueueConnection();
        QueueSession session = connection.createQueueSession(false,
            Session.AUTO_ACKNOWLEDGE);
        QueueSender queueSender = session.createSender(queue);
        TextMessage message = session.createTextMessage();
        message.setText(xmlDoc);
        queueSender.send(message);
        /** Generate and return a correlation identifier ***/
        return generateCorrelationID();
    }
}

```

In Figure 3.40, the vertical lines represent the passage of time, from top to bottom. The vertical rectangular boxes indicate when the entity (client or service) is busy processing the request or waiting for the other entity to complete processing. The half arrow type indicates asynchronous communication and the dashed vertical line indicates that the entity is free to work on other things while a request is being processed.

One question remains: How does the client get the final result of its request? The service may make the result of the client's request available in one of two ways:

- The client that invoked the service periodically checks the status of the request using the correlation identifier that was provided at the time the request was submitted. This is also known as polling, and it appears as Option 1 in Figure 3.40.
- Or, if the client itself is a Web service peer, the service calls back the client's service with the result. The client may use the correlation identifier to relate the response with the original request (Option 2 in Figure 3.40).

Often this is decided by the nature of the service itself. For example, if the service runs a business process workflow, the workflow requires the service to take appropriate action after processing the request.

**Formulating Responses.** After you delegate the request to the business logic portion of the application, and the business logic completes its processing, you are ready for the next step: to form the response to the request.

Consider response generation from the weather information service's point-of-view. The weather information service may be used by a variety of client types, from browsers to rich clients to handheld devices. A well-designed weather information service would render its responses in formats suitable for these different client types.

However, it is not good design to have a different implementation of the service's logic for each client type. Rather, it is better to design a common business logic for all client types. Then, in the interaction layer, transform the results per client type for rendering. It is thus important to consider the above guidelines, especially when your service has a common processing logic but potentially has different response rendering needs to fit its varied client types.

### 3.3.10.11 Publishing a Web Service

Your Web service needs to be accessible to its intended clients. Recall that some Web services are intended for use by the general public. Other Web services are intended to be used only between trusted business partners (inter-enterprise), and still others are intended for use just within an enterprise (intra-enterprise).

Regardless of whether a service is to be accessible to the public, other enterprises, or even within a single enterprise, you must first make the details about the Web service—its interface, parameters, where the service is located, and so forth—accessible to clients. You do so by making a description of the Web service available to interested parties. WSDL is the standard language for describing a service. Making this WSDL description available to clients enables them to use the service.

Once the WSDL is ready, you have the option to publish it in a registry. The next section describes when you might want to publish the WSDL in a registry. If you make the WSDL description of your service available in a public registry, then a Java-based client can use the JAXR APIs to look up the description of your service and then use the service. For that matter, a client can use the same JAXR APIs to look up the description of any Web service with an available WSDL description. This section examines registries from the point of view of a service developer.

**Publishing a service in a registry** is one method of making the service available to clients. If you decide to publish your service in a registry, you decide on the type of registry to use based on the likely usage scenarios for your service. Registries run the gamut from public registries to corporate registries available only within a single enterprise.

You may want to register Web services for general public consumption on a well-known public registry.

When you make your service available through a public registry, you essentially open the service's accessibility to the widest possible audience. When a service is registered in a public registry, any client, even one with no prior knowledge of the service, may look up and use the service. Keep in mind that the public registry holds the Web service description, which consists not only of the service's WSDL description but also any XML schemas referenced by the service description. In short, your Web service must publish its public XML schemas and any additional schemas defined in the context of the service. You also must publish on the same public registry XML schemas referred to by the Web service description.

When a Web service is strictly for intra-enterprise use, you may publish a Web service description on a corporate registry within the enterprise.

You do not need to use a registry if all the customers of your Web services are dedicated partners and there is an agreement among the partners on the use of the services. When this is the case, you can publish your Web service description—the WSDL and referenced XML schemas—at a well-known location with the proper access protections.

**Understanding Registry Concepts.** When considering whether to publish your service via a registry, it is important to understand some of the concepts, such as repositories and taxonomies, that are associated with registries.

Public registries are not repositories. Rather than containing complete details on services, public registries contain only details about what services are available and how to access these services. For example, a service selling adventure packages cannot register its complete catalog of products. A registry can only store the type of service, its location, and information required to access the service. A client interested in a service must first discover the service from the registry and then bind with the service to obtain the service's complete catalog of products. Once it obtains the service's catalog, the client can ascertain whether the particular service meets its needs. If not, the client must go back to the registry and repeat the discovery and binding process—the client looks in the registry for some other service that potentially offers what it wants, binds to that service, obtains and assesses its catalog, and so forth. Since this process, which is not insignificant, may have to be repeated several times, it is easy to see that it is important to register a service under its proper taxonomy.

Register a service under the proper taxonomy. It is important to register your service under the proper taxonomies. When you want to publish your service on a registry, either a public or corporate registry, you must do so against a taxonomy that correctly classifies or categorizes your Web service. It is important to decide on the proper taxonomy, as this affects the ease with which clients can find and use your service. Several well-defined industry standard taxonomies exist today, such as those defined by organizations such as the North American Industry Classification System (NAICS).

Using existing, well-known taxonomies gives clients of your Web service a standard base from which to search for your service, making it easy for clients to find your service. For example, suppose your travel business provides South Sea island-related adventure packages as well as alpine or mountaineering adventures. Rather than create your own taxonomy to categorize your service, clients can more easily find your service if you publish your service description using two different standard taxonomies: one taxonomy for island adventures and another for alpine and mountaineering adventures.

You can publish your Web service in more than one registry. To further help clients find your service, it is also a good idea to publish in as many applicable categories as possible. For example, a travel business selling adventure packages might register using a product category taxonomy as well as a geographical taxonomy. This gives clients a chance to use optimal strategies for locating a service. For example, if multiple instances of a service exist for a particular product, the client might further refine its selection by considering geographical location and choosing a service close to its own location. Using the travel business service as an example, such a service might register under the taxonomies for types of adventure packages (island and mountaineering), as well as under the taxonomies for the locales in which the adventure packages are provided (Mount Kilimanjaro or Tahiti), thus making it as easy as possible for a prospective client to locate its services.

**Registry Implementation Scenarios.** Once you decide to publish your service and establish the taxonomies that best identify your service, you are ready to implement your decisions. Before doing so, you may find it helpful to examine some of the registry implementation scenarios that you may encounter.

When a registry is used, we have seen that the service provider publishes the Web service description on a registry and clients discover and bind to the Web service to use its services. In general, a client must perform three steps to use a Web service:

1. The client must determine how to access the service's methods, such as determining the service method parameters, return values, and so forth. This is referred to as discovering the service definition interface.
2. The client must locate the actual Web service; that is, find the service's address. This is referred to as discovering the service implementation.
3. The client must be bound to the service's specific location, and this may occur on one of three occasions:

- When the client is developed (called static binding)
- When the client is deployed (also called static binding)
- During runtime (called dynamic binding)

These three steps may produce three scenarios. The particular scenario depends on when the binding occurs and whether the client is implemented solely for a specific service or is a generic client. The following paragraphs describe these scenarios. (See Table 3.14 for a summary.) They also note important points you should consider when designing and implementing a Web service.

- Scenario 1: The Web service has an agreement with its partners and publishes its WSDL description and referenced XML schemas at a well-known, specified location. It expects its client developers to know this location. When this is the case, the client is implemented with the service's interface in mind. When it is built, the client is already designed to look up the service interface directly rather than using a registry to find the service.
- Scenario 2: Similar to scenario 1, the Web service publishes its WSDL description and XML schemas at a well-known location, and it expects its partners to either know this location or be able to discover it easily. Or, when the partner is built, it can use a tool to dynamically discover and then include either the service's specific implementation or the service's interface definition, along with its specific implementation. In this case, binding is static because the partner is built when the service interface definition and implementation are already known to it, even though this information was found dynamically.
- Scenario 3: The service implements an interface at a well-known location, or it expects its clients to use tools to find the interface at build time. Since the Web service's clients are generic clients—they are not clients designed solely to use this Web service—you must design the service so that it can be registered in a registry. Such generic clients dynamically find a service's specific implementation at runtime using registries. Choose the type of registry for the service—either public, corporate, or private—depending on the types of its clients—either general public or intra-enterprise—its security constraints, and so forth.

**Table 3.14. Discovery-Binding Scenarios for Clients**

Scenarios	Discover Service Interface Definition	Discover Service Implementation	Binding to Specific Location
1	None	None	Static
2	None or dynamic at build time	Dynamic at build time	Static
3	None or dynamic at build time	Dynamic at runtime	Dynamic at build time

### 3.3.10.12 Handling XML Documents in a Web Service

Up to now, this chapter addressed issues applicable to all Web service implementations. There are additional considerations when a Web service implementation expects to receive an XML document containing all the information from a client, and which the service uses to start a business process to handle the request. There are several reasons why it is appropriate to exchange documents:

- Documents, especially business documents, may be very large, and as such, they are often sent as a batch of related information. They may be compressed independently from the SOAP message.
- Documents may be legally binding business documents. At a minimum, their original form needs to be conserved through the exchange and, more than likely, they may need to be archived and kept as evidence in case of disagreement. For these documents, the complete infoset of the original document should be preserved, including comments and external entity references (as well as the referred entities).
- Some application processing requires the complete document infoset, including comments and external entity references. As with the legally binding documents, it is necessary to preserve the complete infoset, including comments and external entity references, of the original document.
- When sent as attachments, it is possible to handle documents that may conform to schemas expressed in languages not supported by the Web service endpoint or that are prohibited from being present within a SOAP message infoset (such as the Document Type Declaration `<!DOCTYPE>` for a DTD-based schema).

For example, consider the travel agency Web service, which typically receives a client request as an XML document containing all information needed to arrange a particular trip. The information in the document includes details about the customer's account, credit card status, desired travel destinations, preferred airlines, class of

travel, dates, and so forth. The Web service uses the documents contents to perform such steps as verifying the customer's account, obtaining authorization for the credit card, checking accommodations and transportation availability, building an itinerary, and purchasing tickets.

In essence, the service, which receives the request with the XML document, starts a business process to perform a series of steps to complete the request. The contents of the XML document are used throughout the business process. Handling this type of scenario effectively requires some considerations in addition to the general ones for all Web services.

Good design expects XML documents to be received as `javax.xml.transform.Source` objects. It is good design to do the validation and any required transformation of the XML documents as close to the endpoint as possible. Validation and transformation should be done before applying any processing logic to the document content.

It is important to consider the processing time for a request and whether the client waits for the response. When a service expects an XML document as input and starts a lengthy business process based on the document contents, then clients typically do not want to wait for the response. Good design when processing time may be extensive is to delegate a request to a JMS queue or topic and return a correlation identifier for the client's future reference.

**Exchanging XML Documents.** The J2EE platform provides three ways to exchange XML documents. The first option is to use the Java-MIME mappings provided by the J2EE platform. With this option, the Web service endpoint receives documents as `javax.xml.transform.Source` objects. Along with the document, the service endpoint can also expect to receive other JAX-RPC arguments containing metadata, processing requirements, security information, and so forth. When an XML document is passed as a `Source` object, the container automatically handles the document as an attachment—effectively, the container implementation handles the document-passing details for you. This frees you from the intricacies of sending and retrieving documents as part of the endpoint's request/response handling.

Passing XML documents as `Source` objects is the most effective option in a completely Java-based environment (one in which all Web service clients are based on Java). However, sending documents as `Source` objects may not be interoperable with non-Java clients.

The second option is to design your service endpoint such that it receives documents as `String` types. Code Example 3.53 shows the WSDL description for a service that receives documents as `String` types, illustrating how the WSDL maps the XML document.

#### *Example 3.53. Mapping XML Document to xsd:string*

```
<?xml version="1.0" encoding="UTF-8"?>
<definitions ...>
    <types/>
    <message name="PurchaseOrderService_submitPurchaseOrder">
        <part name="PurchaseOrderXMLDoc" type="xsd:string"/>
    </message>
    <message
        name="PurchaseOrderService_submitPurchaseOrderResponse">
        <part name="result" type="xsd:string"/>
    </message>
    <portType name="PurchaseOrderService">
        <operation name="submitPurchaseOrder"
            parameterOrder="PurchaseOrderXMLDoc">
            <input
                message="tns:PurchaseOrderService_submitPurchaseOrder"/>
            <output message=
                "tns:PurchaseOrderService_submitPurchaseOrderResponse"/>
        </operation>
    </portType>
    ...
</definitions>
```

Code Example 3.54 shows the equivalent Java interface for the WSDL shown in Code Example 3.53.

#### **Example 3.54. Receiving an XML Document as a String object**

```
public interface PurchaseOrderService extends Remote {  
    public String submitPurchaseOrder(String poDocument)  
        throws RemoteException, InvalidOrderException; }
```

If you are developing your service using the Java-to-WSDL approach, and the service must exchange XML documents and be interoperable with clients on any platform, then passing documents as String objects may be your only option.

There may be a performance drawback to sending an XML document as a String object: As the document size grows, the String equivalent size of the document grows as well. As a result, the payload size of the message you send also grows. In addition, the XML document loses its original format since sending a document as a String object sends it in a canonical format.

The third option is to exchange the XML document as a SOAP document fragment. With this option, you map the XML document to xsd:anyType in the service's WSDL file. It is recommended that Web services exchange XML documents as SOAP document fragments because passing XML documents in this manner is both portable across J2EE implementations and interoperable with all platforms.

To pass SOAP document fragments, you must implement your service using the WSDL-to-Java approach.

For example, the travel agency service receives an XML document representing a purchase order that contains all details about the customer's preferred travel plans. To implement this service, you define the WSDL for the service and, in the WSDL, you map the XML document type as xsd:anyType. See Code Example 3.55.

#### **Example 3.55. Mapping XML document to xsd:anyType**

```
<?xml version="1.0" encoding="UTF-8"?>  
<definitions ...>  
    <types/>  
    <message name="PurchaseOrderService_submitPurchaseOrder">  
        <part name="PurchaseOrderXMLDoc" type="xsd:anyType"/>  
    </message>  
    <message  
        name="PurchaseOrderService_submitPurchaseOrderResponse">  
        <part name="result" type="xsd:string"/>  
    </message>  
    <portType name="PurchaseOrderService">  
        <operation name="submitPurchaseOrder"  
            parameterOrder="PurchaseOrderXMLDoc">  
            <input  
                message="tns:PurchaseOrderService_submitPurchaseOrder"/>  
            <output message=  
                "tns:PurchaseOrderService_submitPurchaseOrderResponse"/>  
        </operation>  
    </portType>  
    ...  
</definitions>
```

A WSDL mapping of the XML document type to xsd:anyType requires the platform to map the document parameter as a javax.xml.soap.SOAPElement object. For example, Code Example 3.56 shows the Java interface generated for the WSDL description in Code Example 3.55.

#### **Example 3.56. Java Interface for WSDL in Code Example DES.E31**

```
public interface PurchaseOrderService extends Remote {  
    public String submitPurchaseOrder(SOAPElement  
        purchaseOrderXMLDoc) throws RemoteException; }
```

In this example, the `SOAPElement` parameter in `submitPurchaseOrder` represents the SOAP document fragment sent by the client. For the travel agency service, this is the purchase order. The service can parse the received SOAP document fragment using the `javax.xml.soap.SOAPElement` API. Or, the service can use JAXB to map the document fragment to a Java Object or transform it to another schema. A client of this Web service builds the purchase order document using the client platform-specific API for building SOAP document fragments—on the Java platform, this is the `javax.xml.soap.SOAPElement` API—and sends the document as one of the Web service's call parameters.

When using the WSDL-to-Java approach, you can directly map the document to be exchanged to its appropriate schema in the WSDL. The corresponding generated Java interface represents the document as its equivalent Java Object. As a result, the service endpoint never sees the document that is exchanged in its original document form. It also means that the endpoint is tightly coupled to the document's schema: Any change in the document's schema requires a corresponding change to the endpoint. If you do not want such tight coupling, consider using `xsd:anyType` to map the document.

**Using JAXM and SAAJ Technologies.** The J2EE platform provides an array of technologies—including mandatory technologies such as JAX-RPC and SAAJ and optional technologies such as Java API for XML Messaging (JAXM)—that enable message and document exchanges with SOAP. Each of these J2EE technologies offers a different level of support for SOAP-based messaging and communication.

- SAAJ lets developers deal directly with SOAP messages, and is best suited for point-to-point messaging environments. SAAJ is better for developers who want more control over the SOAP messages being exchanged and for developers using handlers.
- JAXM defines an infrastructure for guaranteed delivery of messages. It provides a way of sending and receiving XML documents and guaranteeing their receipt, and is designed for use cases that involve storing and forwarding XML documents and messages.

SAAJ is considered more useful for advanced developers who thoroughly know the technology and who must deal directly with SOAP messages.

Using JAXM for scenarios that require passing XML documents may be a good choice. Note, though, that JAXM is optional in the J2EE platform. As a result, a service developed with JAXM may not be portable. When you control both end points of a Web service, it may make more sense to consider using JAXM.

### 3.3.10.13 Deploying and Packaging a Service Endpoint

Up to now, we have examined Web services on the J2EE platform in terms of design, development, and implementation. Once you complete the Web services implementation, you must write its deployment descriptors, package the service with all its components, and deploy the service.

Developers should, if at all possible, use tools or IDEs to develop a Web service. These Web service development tools and IDEs automatically create the proper deployment descriptors for the service and correctly handle the packaging of the service—steps necessary for a service to operate properly. Furthermore, tools and IDEs hide these details from the developer.

Although you can expect your development tool to perform these tasks for you, it is good to have a conceptual understanding of the J2EE platform deployment descriptor and packaging structure, since they determine how a service is deployed on a J2EE server and the service's availability to clients. This section, which provides a conceptual overview of the deployment and packaging details, is not essential reading. Nonetheless, you may find it worthwhile to see how these details contribute to portable, interoperable Web services.

**Service Information in the deployment Descriptors.** To successfully deploy a service, the developer provides the following information.

- Deployment-related details of the service implementation, including the Web service interface, the classes that implement the Web service interface, and so forth.
- Details about the Web services to be deployed, such as the ports and mappings
- Details on the WSDL port-to-port component relationship

More specifically, the deployment descriptor contains information about a service's port and associated WSDL:

- A port component (also called a port) gives a view of the service to clients such that the client need not worry about how the service has been implemented.
- Each port has an associated WSDL.
- Each port has an associated service endpoint (and its implementation). The endpoint services all requests that pass through the location defined in the WSDL port address.

To begin, the service implementation declares its deployment details in the appropriate module-specific deployment descriptors. For example, a service implementation that uses a JAX-RPC service endpoint declares its details in the WEB-INF/web.xml file using the servlet-class element. (See Code Example 3.57)

**Example 3.57 web.xml File for a JAX-RPC Service Endpoint**

```
<web-app ...>
  ...
  <servlet>
    <description>Endpoint for Some Web Service</description>
    <display-name>SomeWebService</display-name>
    <servlet-name>SomeService</servlet-name>
    <servlet-class>com.a.b.c.SomeServiceImpl</servlet-class>
    <load-on-startup>0</load-on-startup>
  </servlet>
  <servlet-mapping>
    <servlet-name>SomeService</servlet-name>
    <url-pattern>/webservice/SomeService</url-pattern>
  </servlet-mapping>
  ...
</web-app>
```

Note that when you have a service that functions purely as a Web service using JAX-RPC service endpoints, some specifications in the web.xml file, such as <error-page> and <welcome-file-list>, have no effect.

A service implementation that uses an EJB service endpoint declares its deployment details in the file META-INF/ejb-jar.xml using the session element. (See Code Example 3.58)

**Example 3.58. ejb-jar.xml File for an EJB Service Endpoint**

```
<ejb-jar ...>
  <display-name>Some Enterprise Bean</display-name>
  <enterprise-beans>
    <session>
      <ejb-name>SomeBean</ejb-name>
      <service-endpoint>com.a.b.c.SomeIntf</service-endpoint>
      <ejb-class>com.a.b.c.SomeServiceEJB</ejb-class>
      <session-type>Stateless</session-type>
      <transaction-type>Container</transaction-type>
    </session>
  </enterprise-beans>
  ...
</ejb-jar>
```

Next, the details of the port are specified. The Web service deployment descriptor, called webservices.xml, defines and declares the structural details for the port of a Web service. This file contains the following information:

- A logical name for the port that is also unique among all port components (port-component-name element)
- The service endpoint interface for the port (service-endpoint-interface element)
- The name of the class that implements the service interface (service-impl-bean element)
- The WSDL file for the service (wsdl-file element)
- A QName for the port (wsdl-port element)
- A correlation between WSDL definitions and actual Java interfaces and definitions using the mapping file (jaxrpc-mapping-file element)
- Optional details on any handlers

The reference to the service implementation bean, specified using the service-impl-bean element in webservices.xml, is either a servlet-link or an ejb-link depending on whether the endpoint is a JAX-RPC or EJB service endpoint. This link element associates the Web service port to the actual endpoint implementation defined in either the web.xml or ejb-jar.xml file.

The JAX-RPC mapping file, which is specified using the jaxrpc-mapping-file element in webservices.xml, keeps details on the relationships and mappings between WSDL definitions and corresponding Java interfaces and definitions. The information contained in this file, along with information in the WSDL, is used to create stubs and ties for deployed services.

Thus, the Web services deployment descriptor, webservices.xml, links the WSDL port information to a unique port component and from there to the actual implementation classes and Java-to-WSDL mappings. Code Example 3.59 is an example of the Web services deployment descriptor for our sample weather Web service, which uses a JAX-RPC service endpoint.

#### **Example 3.59. Weather Web Service Deployment Descriptor**

```
<webservices ...>
    <description>Web Service Descriptor for weather service
    </description>
    <webservice-description>
        <webservice-description-name>
            WeatherWebService
        </webservice-description-name>
        <wsdl-file>
            WEB-INF/wsdl/WeatherWebService.wsdl
        </wsdl-file>
        <jaxrpc-mapping-file>
            WEB-INF/WeatherWebServiceMapping.xml
        </jaxrpc-mapping-file>
        <port-component>
            <description>port component description</description>
            <port-component-name>
                WeatherServicePort
            </port-component-name>
            <wsdl-port xmlns:weatherns="urn:WeatherWebService">
                weatherns:WeatherServicePort
            </wsdl-port>
            <service-endpoint-interface>
                endpoint.WeatherService
            </service-endpoint-interface>
            <service-impl-bean>
                <servlet-link>WeatherService</servlet-link>
            </service-impl-bean>
        </port-component>
    </webservice-description>
</webservices>
```

**Package Structure.** Once the service implementation and deployment descriptors are completed, the following files should be packaged into the appropriate J2EE module:

- The WSDL file
- The service endpoint interface, including its implementation and dependent classes
- The JAX-RPC mapping file, which specifies the package name containing the generated runtime classes and defines the namespace URI for the service.
- The Web service deployment descriptor

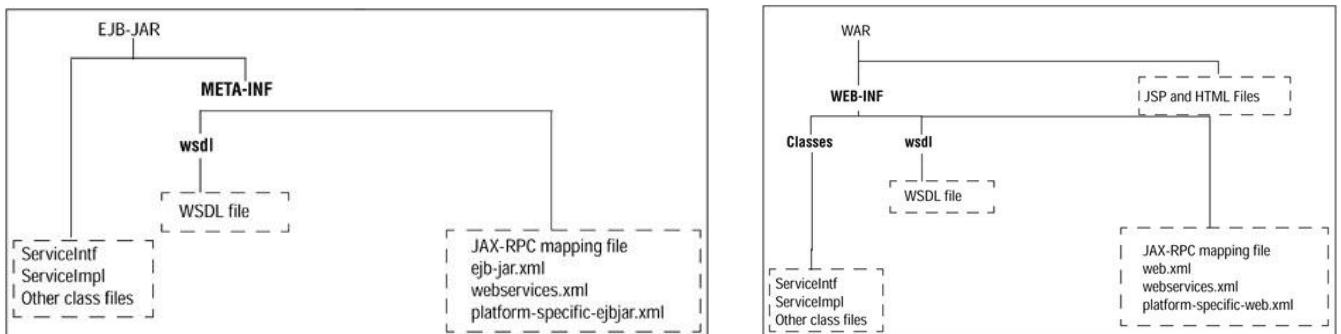
The type of endpoint used for the service implementation determines the type of the J2EE module to use.

The appropriate J2EE module for a service with a JAX-RPC service endpoint is a WAR file. A service using an EJB service endpoint must be packaged in an EJB-JAR file.

The package structure is as follows:

- WSDL files are located relative to the root of the module.
- The service interface, the service implementation classes, and the dependent classes are packaged just like any other J2EE component.
- The JAX-RPC mapping file is located relative to the root of the module (typically in the same place as the module's deployment descriptor).
- The Web service deployment descriptor location depends on the type of service endpoint, as follows:
  - For an EJB service endpoint, the Web service deployment descriptor is packaged in an EJB-JAR in the META-INF directory as META-INF/webservices.xml
  - For a JAX-RPC service endpoint, the deployment descriptor is packaged in a WAR file in the WEB-INF directory as WEB-INF/webservices.xml.

See Figure 3.41(a), which shows a typical package structure for a Web service using an EJB endpoint. Figure 3.41(b) shows the typical structure for a Web service using a JAX-RPC endpoint.



**Fig. 3.41 (a)Package Structure for EJB Endpoint (b) Package Structure for JAX-RPC Service Endpoint**

### 3.3.10.14 Client Design

Web services take the Web client model to the next level. Developers can write far more powerful clients whose interaction with Web services provides a rich client experience. In this environment, client developers need not have control over the server portion of an application, yet they can still write powerful, rich client applications. This chapter focuses on using the Java platform to design and develop Web services-based clients.

Clients can take advantage of Web services to obtain a wide range of functions or services. To a client, a Web service is a black box: The client does not have to know how the service is implemented or even who provides it. The client primarily cares about the functionality—the service—provided by the Web service. Examples of Web services include order tracking services, information look-up services, and credit card validation services. Various clients running on different types of platforms can all access these Web services.

One of the principal reasons for implementing Web services is to achieve interoperability. Clients can access Web services regardless of the platform or operating system upon which the service is implemented. Not only is the service's platform of no concern to the client, the client's implementation language is completely independent of the service.

Web service clients can take many forms, from full-blown J2EE applications, to rich client applications, even to light-weight application clients, such as wireless devices. In short, there are many different types of clients that can talk to Web services. The Java platform provides excellent support for writing Web service clients. Web services also provide clients a standardized approach to access services through firewalls. Such access extends the capabilities of clients. Clients accessing Web services also remain more loosely coupled to the service.

#### Choosing a Communication Technology

Web services are only one of several ways for a client to access an application service. For example, Java applications may access application services using RMI/IOP, JMS, or Web services. There are advantages and disadvantages with each of these communication technologies, and the developer must weigh these considerations when deciding on the client application design.

Interoperability is the primary advantage for using Web services as a means of communication. Web services give clients the ability to interoperate with almost any type of system and application, regardless of the platform on which the system or application runs. In addition, the client can use a variety of technologies for this communication. Furthermore, different client types—such as handheld devices, desktop browsers, or rich GUI clients—running on different platforms and written in different languages may be able to access the same set of services. Some applications may be designed such that their functionality is only accessible via a Web service.

Web services use HTTP as the transport protocol, which enables clients to operate with systems through firewalls. The service's WSDL document enables clients and services that use very different technologies to map and convert their respective data objects. For services and clients that are based on JAX-RPC, the JAX-RPC runtime handles this mapping transparently.

Let's look at the different communication approaches that are available to a J2EE client to access a service, including Web services, RMI/IOP, and Java Message Service. Clients can easily use the JAX-RPC-generated stub classes to access a Web service. Although not as fast from a performance perspective as other technologies (such as RMI/IOP), JAX-RPC gives clients greater flexibility and supports more types of clients.

J2EE application clients may also use RMI/IOP to make remote calls over the network on application business logic. RMI/IOP is often used for clients operating in intranet environments, where there is a greater degree of control over the client's deployment and the J2EE server. While these controlled environments provide a client container that handles the communication security, passing through firewalls can be problematic. RMI/IOP provides clients with secure access to the application business logic while at the same time taking care of the details of the client and server communication and marshalling and demarshalling parameters.

Java Message Service (JMS) is another means for J2EE clients to communicate with server applications. JMS provides a means for asynchronous communication. Applications using JMS are better suited to a setting that is behind a firewall, since messaging systems generally do not work well on the Internet. (Often, messaging systems are not even exposed on the Internet.) Not only must developers have some knowledge of how to work with messaging systems, such as how to set up and use topics or queues, but the messaging system mechanisms must already be in place.

Although Web services provide a standard way to exchange XML documents over HTTP, you can use nonstandard approaches as well. Communication using the HTTP protocol requires only a simple infrastructure to send and receive messages. However, the client application must be able to parse the XML documents representing the messages. Parsing involves mapping the XML data to the client application's object model. When using this means of communication, the developer at a minimum needs to write code to send and receive the documents over HTTP as well as to parse the document data. If such communication must also be secure, developers would have to include code that uses Secure Socket Layer (SSL), making the development task more difficult. However, this means of communication may be sufficient, particularly in a closed environment or when clients are applets. Care should be taken if using this approach, since it is not standard.

## Scenarios for Web Services-Based Client Applications

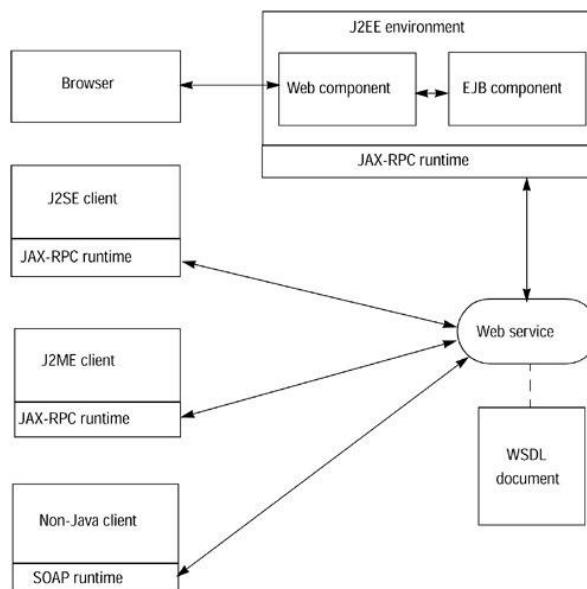
Developers typically write clients to access preexisting Web services—that is, public or private Web services. At times, the same developers may simultaneously develop both the service and its clients. Regardless of this, client developers should rely on the WSDL service description for their knowledge of the service details. In addition, developers may use a variety of technologies and APIs for developing these clients. They may develop clients using J2EE technologies, often called J2EE clients, or they may use the standard Java (J2SE and J2ME) technologies, or even non-Java technologies.

Before delving into the details of client development, let's examine several Web service client scenarios. Although different, each scenario is based on accessing the same Web service. The scenarios are as follows:

- J2EE component— In this scenario, a J2EE component accesses a Web service. The J2EE component receives results from the service, and it formats these results so that it can be read or displayed by a browser.
- J2SE client— A J2SE client may access the same Web service as the J2EE component. However, the J2SE client provides a more detailed view of the results returned from the service.

- J2ME client— A J2ME client, such as a client application running on a mobile device or PDA, gives a user the freedom to access the same Web service from places other than his or her office. In addition, the user can work offline with the results returned from the service.
- Non-Java client— A non-Java client accesses the same Web service using SOAP over HTTP.

Figure 3.42 shows how these different types of clients might access the same purchase order tracking Web service interface. All clients, regardless of their platform, rely on the Web service's WSDL document, which uses a standard format to describe the service's location and its operations. Clients need to know the information in the WSDL document—such as the URL of the service, types of parameters, and port names—to understand how to communicate with a particular service.



**Fig. 3.42. Web Service Clients in J2EE and Non-J2EE Environments**

It is important to note that none of the clients communicate directly with a Web service. Each client type relies on a runtime—either a JAX-RPC or SOAP runtime—through which it accesses the service. From the developer's perspective, the client's use of a runtime is kept nearly transparent. However, good design dictates that a developer still modularize the Web service access code, plus consider issues related to remote calls and handling remote exceptions.

Browser-based applications rely on J2EE components, which act as clients of the Web service. These clients, referred to as J2EE clients, are J2EE components that access a Web service, and the JAX-RPC runtime in turn handles the communication with the Web service. The J2EE environment shields the developer from the communication details. J2EE clients run in either an EJB container or a Web container, and these containers manage the client environment.

Stand-alone clients—which may be J2SE clients, J2ME clients, or clients written in a language other than Java—communicate to the Web service through the JAX-RPC runtime or the SOAP runtime. Stand-alone clients are outside the J2EE environment. Because they don't have a J2EE EJB container or Web container to manage the environment, stand-alone clients require more work from the developer.

Although each type of client works well, developers must deal with an increasing level of complexity if they work directly with the JAX-RPC or SOAP runtimes. The advantage of the J2EE environment is that it shields developers from some of the complexity associated with developing Web services, such as the look up of the service and the life-cycle management of objects used to access a service. In the J2EE environment, a developer uses JNDI to look up a service in much the same way as he or she might use other Java APIs, such as JDBC or JMS.

Given that many types of clients can access Web services, how do you determine which type of client is best for your application? The following general guidelines should help with this decision:

**J2EE clients**— J2EE clients have good access to Web services. J2EE clients have other advantages provided by the J2EE platform, such as declarative security, transactions, and instance management. J2EE clients may also access Web services from within a workflow architecture, and they may aggregate Web services.

**J2SE clients**— Generally, J2SE clients are best when you need to provide a rich interface or when you must manipulate large sets of data. J2SE clients may also work in a disconnected mode of operation.

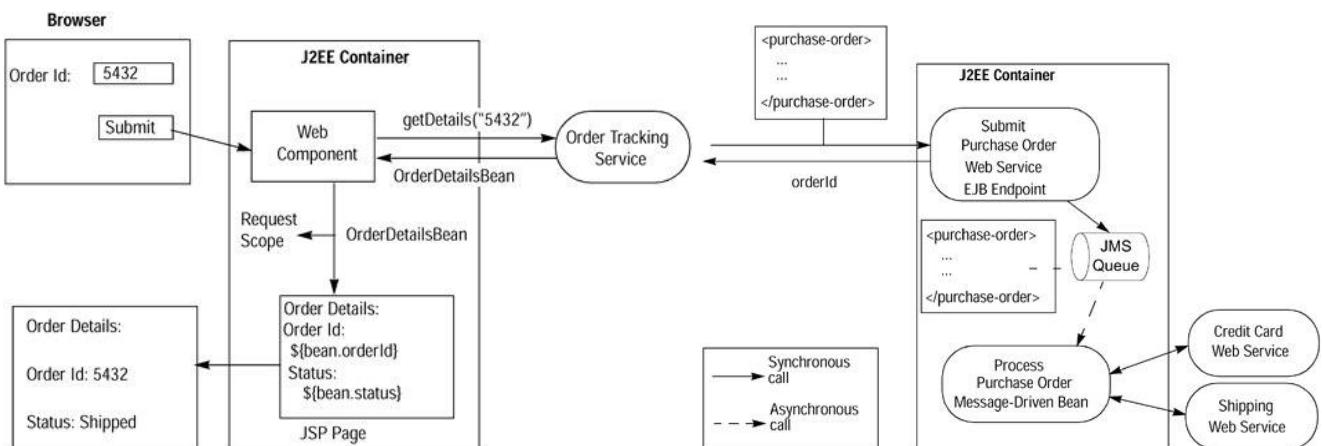
**J2ME clients**— J2ME clients are best for applications that require remote and immediate access to a Web service. J2ME clients may be restricted to a limited set of interface components. Like J2SE clients, J2ME clients may also work in a disconnected mode of operation.

## Designing J2EE Clients

As already noted, the J2EE platform provides an environment that supports client application access to Web services. In a J2EE environment the deployment descriptors declaratively define the client-side Web service configuration information. The deployer may change this configuration information at deployment. In addition, the J2EE platform handles the underlying work for creating and initializing the access to the Web services.

The J2EE platform provides many other technologies in addition to supporting Web services. Developers can obtain a richer set of functionality by using these other services, such as Web and EJB components, in addition to Web services for their client applications.

For example, consider a browser client that accesses an order tracking Web service via a Web component. (See Figure 3.43(a)) The Web component presents a browser page to the end user, who enters an order identifier to a form displayed in its browser. The browser client passes this order identifier to a Web component, which accesses the order tracking Web service and retrieves the order information. The Web component converts the retrieved information to an HTML page and returns the formatted page to the browser client for presentation to the end user.



**Fig.3.43 (a) Web Tier Component Calling a Web Service (b)EJB Components and Web Services in a Workflow**

In this example, a Web component calls the order tracking service and, when it receives a response from the service, it puts the results in a Java Beans component (OrderDetailsBean) that is within the request scope. The Web component uses a JSP to generate an HTML response, which the container returns to the browser that made the original request.

It is also possible to write J2EE clients using EJB components. (See Figure 3.43(b)). These EJB components may themselves be Web service endpoints as well as clients of other Web services. Often, EJB components are used in a workflow to provide Web services with the additional support provided by an EJB container—that is, declarative transactional support, declarative security, and life-cycle management.

Figure 3.43(b) demonstrates a workflow scenario: a Web service endpoint used in combination with a message-driven bean component to provide a workflow operation that runs asynchronously once the initial service starts. The Web service endpoint synchronously puts the purchase order in a JMS message queue and returns an orderId

to the calling application. The message-driven bean listens for messages delivered from the JMS queue. When one arrives, the bean retrieves the message and initiates the purchase order processing workflow. The purchase order is processed asynchronously while the Web service receives other purchase orders.

In this example, the workflow consists of three additional stages performed by separate Web services: a credit card charging service, a shipping service, and a service that sends an order confirmation. The message-driven bean aggregates the three workflow stages. Other systems within an organization may provide these services and they may be shared by many applications.

## Designing J2SE Clients

Unlike the J2EE environment, developers of non-J2EE clients are responsible for much of the underlying work to look up a service and to create and maintain instances of classes that access the service. Since they cannot rely on a container, these developers must create and manage their own services and ensure the availability of all runtime environments needed to access the Web services.

J2SE clients, such as desktop applications developed using the Swing API, have the capability to do more processing and state management than other clients. This type of application client can provide a rich GUI application development environment that includes document editing and graphical manipulation. However, there are some points that should be kept in mind when considering J2SE clients:

- Long-running applications— Using J2SE is particularly good for Web service clients that run for extended periods of time. Because these applications run for long periods, developers must consider that both the client and the Web service may need to maintain the state of at least some of the data. It is possible to embed conversational state within each method invocation, if required by the service's use case.
- Using a rich graphical user interface (GUI) for complex data— J2SE clients can provide users with a rich view of data. Such a rich interface might permit a user to navigate and modify offline a large set of data returned by a service. The client can later update the Web service with any changes to the data.
- Requiring only intermittent network access— J2SE client can use Web services without needing to maintain continuous network access, relieving some of the burden on a network. Care must be taken to ensure data consistency between the service and the client.
- Requiring complex computations on the client— J2SE clients are well-suited for performing complex mathematical calculations, as well as operations that update data, and then submitting the results to a service. For example, these clients have better resources for image manipulation, and they can relieve the server of this burden. Thus, J2SE clients can provide a better environment than a service for a user's interaction with data sets.

For example, a J2SE application may contain a rich GUI used to update catalog information. The user can manipulate attributes as well as graphical data using the application's GUI screens. When the user finishes, the application submits new or updated catalog data to the catalog service.

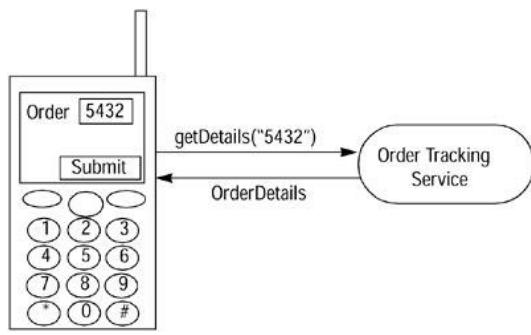
J2SE applications may be deployed using Java Web Start technology. Java Web Start simplifies deployment in large enterprise environments by ensuring that clients have available the proper versions of the Java Runtime Environment and all necessary libraries.

## J2ME Clients

Java 2 Platform, Micro Edition (J2ME) clients may interact remotely with a Web service. However, since J2ME clients have limited GUI capabilities compared to J2SE clients, consider using J2ME clients when mobility and remote access are requirements. Also consider using this type of client when immediacy of access is important.

For example, Figure 3.44 shows a J2ME client running on a cell phone and accessing an order tracking service. The client does not need an elaborate GUI or set of widgets to interact with the Web service.

J2ME clients may access Web services using a subset of the JAX-RPC API. When accessing Web services, J2ME clients should consider network connection and bandwidth issues, offline versus connected operation, and GUI and processing limitations.



**Fig. 3.44. J2ME Client Accessing Web Service**

Networks supporting J2ME devices may not always provide consistent connectivity. Applications using such networks must consider connection failure, or sporadic connectivity, and be designed so that recovery is possible.

Additionally, the bandwidths for many networks supporting J2ME devices limit the rate at which data can be exchanged. Applications need to be designed to limit their data exchange rate to that allowed by the network and to consider the cost that these limitations imply. Care must also be taken to deal with network latency to keep the user experience acceptable. Since Web services do not specify a level of service for message delivery, the client application must take this into account and provide support for message delivery failures.

Keep in mind that J2ME network providers may charge for network usage by the kilobyte. J2ME-targeted applications may be expensive for the user unless care is taken to limit the data transferred.

Applications for J2ME devices may work in an offline, or disconnected, mode as well as an online, or connected, mode. When working in an offline mode, applications should collect data and batch it into requests to the Web service, as well as obtain data from the service in batches. Consideration should be given to the amount of data that is passed between the Web service and the client.

Applications for J2ME devices have a standard, uniform set of GUI widgets available for manipulating data, such as the liquid crystal display UI that is part of MIDP 1.0 and MIDP 2.0. Plus, each device type may have its own set of widgets. Such widgets have different abilities for validating data prior to accessing a service.

J2ME devices, while capable of small computations and data validation, have limited processing capabilities and memory constraints. Client applications need to consider the types of data exchanged and any pre- and post-processing required. For example, J2ME devices have limited support for XML document processing and are not required to perform XML document validation.

### 3.3.10.15 Developing Client Applications to Use a Web Service

All client applications follow certain steps to use a Web service. In brief, they must first look up or locate the service, make a call to the service, and process any returned data. The choice of communication mode determines much of the details for performing these steps.

An application can communicate with a service using stubs, dynamic proxies, or the dynamic invocation interface (DII) Call interface. With these modes, a developer relies on some form of client-side proxy class (stub, dynamic proxy, or a DII interface) representing a Web service to access the service's functionality. The client developer decides which representation to use based on the WSDL availability, the service's endpoint address, and the use cases for the service.

After deciding on a communication mode, developers need to do the following when designing and developing client applications:

1. Assess the nature and availability of the service and the service description.

Typically, the first step is to determine the location of the Web service's WSDL document and choose a communication mode. If you are using stubs or dynamic proxies, you must first locate and gain access to

the full WSDL document representing the Web service, since you develop the client application from the stubs and supporting files generated from the WSDL. If you have access to only a partial WSDL file, then use DII, since DII lets you locate the service at runtime .

2. Create a client-side Web service delegate.
  - o If applicable, generate the necessary stubs and support classes— Generated classes may not be portable across implementations. If portability is important, limit your use of vendor-specific method calls in the stub classes.
  - o Locate the service— Depending on the communication mode, there are different ways to locate a service.
  - o Configure the stub or Call objects.
3. Invoke the service.
  - o When using stubs and dynamic proxies, invocations are synchronous. Use DII if you choose to have a one-way call.
  - o Handle parameters, return values, and exceptions.
4. Present the view.

Often the client may need to generate a view for end users. There are different ways clients can handle the presentation to the user of the Web service response. In some cases, a service sends its response as an XML document, and this document may be transformed or mapped to a suitable format for presentation. Web tier clients might pass the service response to a Web component such as JSP and let the component apply a transformation to the XML document or otherwise handle the returned data. EJB tier clients may themselves apply XSLT transformations to the returned content to create the target content, which they then pass to a Web component.

## Communication Modes for Accessing a Service

There are three principal modes for a client application's communication with a Web service: stub, dynamic proxy, and dynamic invocation interface (DII). By a communication mode, we mean the APIs for programmatically accessing a service via the javax.xml.rpc.Service interface. Clients using either the stubs or dynamic proxies to access a service require the prior definition of a service endpoint interface. Note that when we refer to the service endpoint interface, which is the interface between the stub and dynamic proxy clients and the JAX-RPC API and stub, we are referring to the interface that represents the client's view of a Web service.

When using stubs, a JAX-RPC runtime tool generates during development static stub classes that enable the service and the client to communicate. The stub, which sits between the client and the client representation of the service endpoint interface, is responsible for converting a request from a client to a SOAP message and sending it to the service. The stub also converts responses from the service endpoint, which it receives as SOAP messages, to a format understandable by the client. In a sense, a stub is a local object that acts as a proxy for the service endpoint.

Dynamic proxies provides the same functionality as the stubs, but do so in a more dynamic fashion. Stubs and dynamic proxies both provide the developer access to the javax.xml.rpc.Stub interface, which represents a service endpoint. With both models, it is easy for a developer to program against the service endpoint interface, particularly because the JAX-RPC runtime does much of the communication work behind the scenes. The dynamic proxy model differs from the stub model principally because the dynamic proxy model does not require code generation during development.

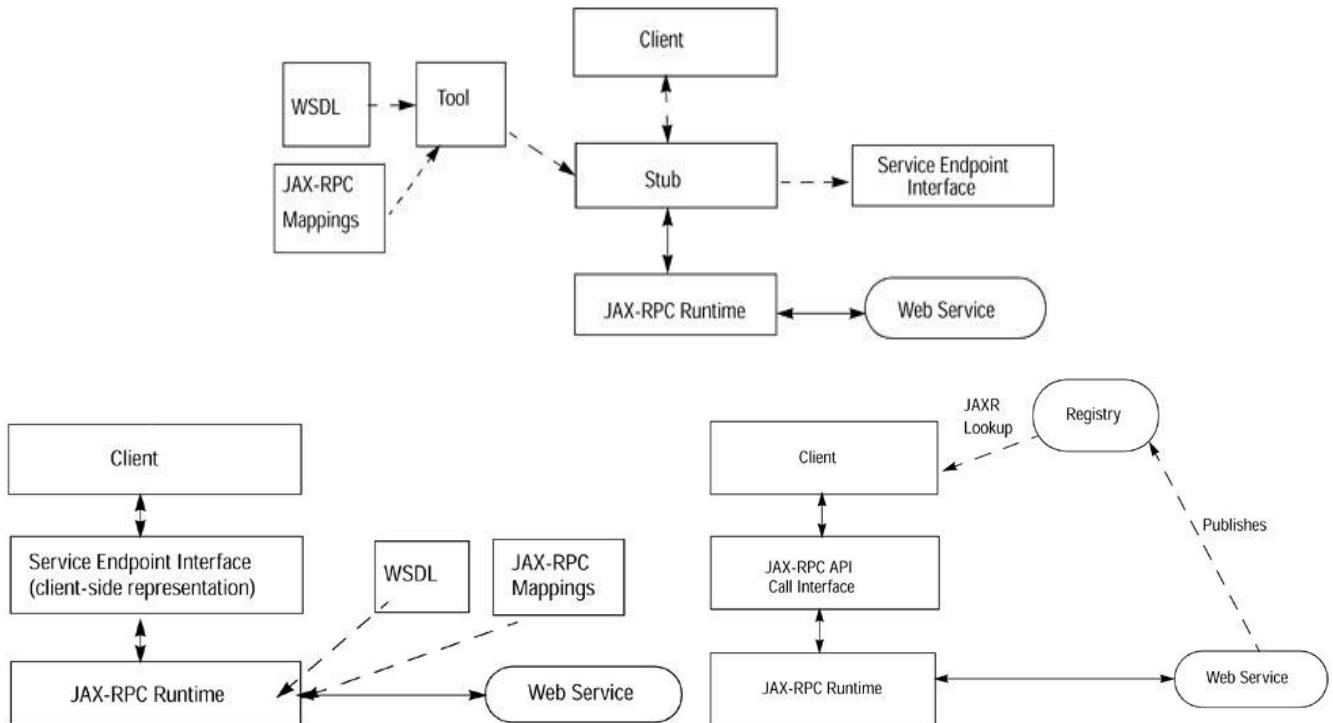
DII is a call interface that supports a programmatic invocation of JAX-RPC requests. Using DII, a client can call a service or a remote procedure on a service without knowing at compile time the exact service name or the procedure's signature. A DII client can discover this information at runtime and can dynamically look up the service and its remote procedures.

When using DII, clients can dynamically access a service at runtime. The stubs rely on a tool that uses the WSDL file to create the service endpoint interface, plus generate stub and other necessary classes. These generated classes eliminate the need for the developer to use the JAX-RPC APIs directly. By contrast, the dynamic proxy and DII approaches both require the developer to use the JAX-RPC APIs.

## Using Stub Communication

J2EE clients should use generated stubs to access services, especially for services with fairly static interfaces. Stub communication easily allows Java access to a Web service. This is the recommended approach for accessing services and their WSDL files when they are unlikely to change over time.

Figure 3.45(a) shows how a client application might access a Web service using a stub generated by a tool prior to the client's deployment and compilation. The WSDL file for the service served as input to the tool. The client-side service endpoint interface, which implements the service endpoint interface on the client-side of the communication channel, provides the client view of the Web service.



**Fig. 3.45.(a)Stub Communication Model (b) Accessing a Service Using a Dynamic Proxy (c) DII Call Interface**

The stub, which implements the client-side service endpoint interface, acts as a proxy to the Web service for the client; that is, the stub is the client's view of the Web service. The tool that generates the stub class also generates all necessary helper classes for accessing the service. These helper classes define the parameters for calls to the service and the service's return values, ensuring that the parameters and return values are all of the proper types expected by the JAX-RPC runtime. To generate these stub and helper classes, the tool relies on the WSDL document as well as a JAX-RPC mapping file. The mapping file supplies information regarding the Java-to-XML bindings, such as the correct package name for generated classes.

The J2ME environment differs somewhat. Applications in J2ME environments can use only stubs to access Web services, and stubs are portable for J2ME devices. The JAX-RPC profile for J2ME environments does not support dynamic proxies and DII.

Although at this time stubs are not portable across J2EE implementations, the next version of JAX-RPC is expected to address this portability issue.

Using stubs, especially in a Java environment, is often the easiest because the developer can work with generated class files representing the service method call parameters and return values. However, the greater dependencies between the client and the Web service may lead to problems if the service interface changes frequently. This mode also requires that stub classes be generated before compiling the application.

## Using Dynamic Proxy Communication

The dynamic proxies are similar in many ways to the stubs previously described. However, unlike stubs, the client developer needs only the client-side interface that matches the service endpoint interface. That is, clients using dynamic proxies program to an interface that ensures the client application is portable across other JAX-RPC runtime implementations. Developers using dynamic proxies must create Java classes to serve as JAX-RPC value types—these classes have an empty constructor and set methods for each field, similar to JavaBeans classes.

The dynamic proxy is based on the service endpoint interface, its WSDL document, and a JAX-RPC mapping file (similar to the stubs model). (See Figure 3.45(b)) Client applications can access the Web service ports using the javax.xml.rpc.Service method getPort.

Note in Figure 3.45(b) that a client accesses a Web service via the client-side representation of the service endpoint interface, while the JAX-RPC runtime handles the work of communicating with the respective service. A developer programs against an interface, which provides a flexible way to access a service in a portable fashion.

Consider using the dynamic proxy approach if portability is important to your application, since this approach uses the service's endpoint interface to communicate with a service at runtime. Dynamic proxy communication is the most portable mode across JAX-RPC implementations.

Because of how they access a service at runtime, dynamic proxies may have additional overhead when calls are made.

## Using DII Call Interface

A client application may also dynamically access a Web service by locating the service at runtime from a registry. The client does not know about the service when it is compiled; instead, the client discovers the service's name from a JAXR registry at runtime. Along with the name, the client discovers the required parameters and return values for making a call to the service. Using a dynamic invocation interface, the client locates the service and calls it at runtime.

Generally, using DII is more difficult for a developer. A developer must work with a more complex interface than with stubs or dynamic proxies. Not only does this interface require more work on the part of the developer, it is more prone to class cast exceptions. In addition, the DII approach may have slower access. A developer may choose to use the DII approach when a complete WSDL document is not available or provided, particularly when the WSDL document does not specify ports. The DII approach is more suitable when used within a framework, since from within a framework, client applications can generically and dynamically access services with no changes to core application code.

Figure 3.45(c) shows how a client uses the JAXR API to look up the endpoint WSDL for a service in a registry. The client uses the information from the registry to construct a javax.xml.rpc.Call, which it uses to access the Web service.

Using the DII Call interface allows a client application to define at runtime the service name and the operations it intends to call on the service, thus giving the client the benefit of loosely coupling its code with that of the service. The client is less affected by changes to the service, whether those changes involve access to the service or data requirements. In addition, the DII communication model permits Web service access code to be standardized among a set of clients and reused as a component.

DII involves more work than stubs or dynamic proxies and should be used sparingly. You may consider using this mode if a service changes frequently.

## Summary of Communication Model Guidelines

Table 3.15 summarizes the communication models for Web service clients.

**Table 3.15 Client Communication Modes**

Client Considerations	Stub	Dynamic Proxy	DII
Portable client code across JAX-RPC implementations	Yes, in the J2EE platform when an application uses a neutral means for accessing the stubs. Since stubs are bound to a specific JAX-RPC runtime, reliance on JAX-RPC-specific access to a stub may not behave the same on all platforms. Stub code needs to be generated for an application.	Yes	Yes
Requires generation of code using a tool	Yes	No. A tool may be used to generate JAX-RPC value types required by a service endpoint interface (but not serializers and other artifacts).	No
Ability to programmatically change the service endpoint URL	Yes, but the WSDL must match that used to generate the stub class and supporting classes.	Yes, but the client-side service endpoint interface must match the representation on the server side.	Yes
Supports service specific exceptions	Yes	Yes	No. All are java.rmi.Remote exceptions. Checked exceptions cannot be used when calls are made dynamically.
Supports one way communication mode	No	No	Yes
Supports the ability to dynamically specify JAX-RPC value types at runtime	No. Developer must program against a service endpoint interface.	No. Developer must program against a service endpoint interface.	Yes. However, returns Java objects which the developer needs to cast to application-specific objects as necessary.
Supported in J2ME platform	Yes	No	No
Supported in J2SE and J2EE platforms	Yes	Yes	Yes
Requires WSDL	No. A service endpoint interface may generate a stub class along with information concerning the protocol binding.	No. A partial WSDL (one with the service port element undefined) may be used.	No. Calls may be used when partial WSDL or no WSDL is specified. Use of methods other than the createCall method on the Call interface may result in unexpected behavior in such cases.

## Locating and Accessing a Service

Locating a service differs depending on the client. Client applications that run in J2EE environments use the JNDI InitialContext.lookup method to locate a service, whereas a J2SE client can use the javax.xml.rpc.ServiceFactory class or an implementation-specific stub to locate a service. Clients use the javax.xml.rpc.Service interface API to access a Web service. A stub implements the service interface.

Code Example 3.60 shows how an application in a J2EE environment might use a stub to access a service. The application locates the service using a JNDI InitialContext.lookup call. The JNDI call returns an OpcOrderTrackingService object, which is a stub.

### **Example 3.60. Accessing a Service with a Stub in a J2EE Environment**

```
Context ic = new InitialContext();
OpcOrderTrackingService opcOrderTrackingSvc = (OpcOrderTrackingService)
    ic.lookup( "java:comp/env/service/OpcOrderTrackingService");
OrderTrackingIntf port = opcOrderTrackingSvc.getOrderTrackingIntfPort();
OrderDetails od = port.getOrderDetails(orderId);
```

Because it depends on the generated stub classes, the client code in Code Example DES.E36 is not the recommended strategy for using stubs. Although this example works without problems, JAX-RPC gives you a neutral way to access a service and obtain the same results. By using the JAX-RPC javax.xml.rpc.Service interface method getPort, you can access a Web service in the same manner regardless of whether you use stubs or dynamic proxies. The getPort method returns either an instance of a generated stub implementation class or a dynamic proxy, and the client can then use this returned instance to invoke operations on the service endpoint.

The getPort method removes the dependencies on generated service-specific implementation classes. When this method is invoked, the JAX-RPC runtime selects a port and protocol binding for communicating with the port, then configures the returned stub that represents the service endpoint interface. Furthermore, since the J2EE platform allows the deployment descriptor to specify multiple ports for a service, the container, based on its configuration, can choose the best available protocol binding and port for the service call. (See Example 3.61)

### **Example 3.61. Looking Up a Port Using a Stub or Dynamic Proxy**

```
Context ic = new InitialContext();
Service service = (Service)ic.lookup(
    "java:comp/env/service/OpcPurchaseOrderService");
PurchaseOrderIntf port=(PurchaseOrderIntf)service.getPort( PurchaseOrderIntf.class);
```

Example 3.61 illustrates how a J2EE client might use the Service interface getPort method. Rather than cast the JNDI reference to the service implementation class, the code casts the JNDI reference to a javax.xml.rpc.Service interface. Using the Service interface in this manner reduces the dependency on generated stub classes. The client developer, by invoking the getPort method, uses the client-side representation of the service endpoint interface to look up the port. After obtaining the port, the client may make any calls desired by the application on the port.

When using stubs or dynamic proxies, the recommended strategy to reduce the dependency on generated classes is to use the java.xml.rpc.Service interface and the getPort method as a proxy for the service implementation class.

A client developer should not circumvent the J2EE platform's management of a service. A client should not create or destroy a Web service port. Instead, a client should use the standard J2EE mechanisms, such as those shown in Code Example 3.61.

A client developer should not assume that the same port instance for the service is used for all calls to the service. Port instances are stateless, and the J2EE platform is not required to return a previously used port instance to a client.

An application in a non-J2EE environment uses a stub to make a Web services call in a different manner. The client application accesses a stub for a service using the method getOrderTrackingIntfPort on the generated implementation class, OpcOrderTrackingService\_Impl, which is specific to each JAX-RPC runtime. J2SE or J2ME clients use these generated \_Impl files because they do not have access to the naming services available to clients in a J2EE environment through JNDI APIs. See Code Example 3.62.

### **Example 3.62. Accessing a Service with a Stub in J2SE and J2ME Environments**

```
Stub stub = (Stub)(new OpcOrderTrackingService_Impl().getOrderTrackingIntfPort());
OrderTrackingIntf port = (OrderTrackingIntf)stub;
```

In addition, a J2SE or J2ME client can access a service by using the javax.xml.rpc.ServiceFactory class to instantiate a stub object. Code Example 3.63 shows how a J2SE client might use a factory to locate the same order tracking service.

### **Example 3.63. Looking Up a Service in J2SE and J2ME Environments**

```
ServiceFactory factory = ServiceFactory.newInstance();
Service service = factory.createService(new QName(
    "urn:OpcOrderTrackingService", "OpcOrderTrackingService"));
```

Similarly, Code Example 3.64 shows how a J2SE application might use a dynamic proxy instead of a stub to access a service.

### **Example 3.64. J2SE Client Dynamic Proxy Service Lookup**

```
ServiceFactory sf = ServiceFactory.newInstance();
String wsdlURI = "http://localhost:8001/webservice/OtEndpointEJB?WSDL";
URL wsdlURL = new URL(wsdlURI);
Service ots = sf.createService(wsdlURL,
    new QName("urn:OpcOrderTrackingService", "OpcOrderTrackingService"));
OrderTrackingIntf port = (
    OrderTrackingIntf)ots.getPort(new QName("urn:OpcOrderTrackingService",
    "OrderTrackingIntfPort"), OrderTrackingIntf.class);
```

Code Example 3.64 illustrates how a J2SE client might program to interfaces that are portable across JAX-RPC runtimes. It shows how a J2SE client uses a ServiceFactory to look up and obtain access to the service, represented as a Service object. The client uses the qualified name, or QName, of the service to obtain the service's port. The WSDL document defines the QName for the service. The client needs to pass as arguments the QName for the target service port and the client-side representation of the service endpoint interface.

By contrast, Code Example 3.65 shows how a J2EE client might use a dynamic proxy to look up and access a service. These two examples show how much simpler it is for J2EE clients to look up and access a service than it is for J2SE clients, since a JNDI lookup from the InitialContext of an existing service is much simpler than configuring the parameters for ServiceFactory. The J2EE client just invokes a getPort call on the client-side representation of the service endpoint interface.

### **Example 3.65. Using a Dynamic Proxy in a J2EE Environment**

```
Context ic = new InitialContext();
Service ots = (Service) ic.lookup("java:comp/env/service/OpcOrderTrackingService");
OrderTrackingIntf port = (OrderTrackingIntf)ots.getPort(
    OrderTrackingIntf.class);
```

A J2SE client using the DII approach might implement the code shown in Code Example 3.66 to look up the same service at runtime. DII communication supports two invocation modes: synchronous and one way, also called fire and forget. Both invocation modes are configured with the javax.xml.rpc.Call object. Note that DII is the only communication model that supports one-way invocation. Code Example 3.66 illustrates using the DII approach for locating and accessing a service. It shows how the Call interface used by DII is configured with the property values required to access the order tracking Web service. The values set for these properties may have been obtained from a registry. Keep in mind that using DII is complex and often requires more work on the part of the client developer.

### **Example 3.66. J2SE Client Using DII to Access a Web Service**

```
Service service = //get service
QName port = new QName("urn:OpcOrderTrackingService", "OrderTrackingIntfPort");
Call call = service.createCall(port);
call.setTargetEndpointAddress( "http://localhost:8000/webservice/OtEndpointEJB");
call.setProperty(Call.SOAPACTION_USE_PROPERTY,new Boolean(true));
call.setProperty(Call.SOAPACTION_URI_PROPERTY,"");
call.setProperty(ENCODING_STYLE_PROPERTY, URI_ENCODING);
QName QNAME_TYPE_STRING = new QName(NS_XSD, "string");
call.setReturnType(QNAME_TYPE_STRING);
call.setOperationName(new QName(BODY_NAMESPACE_VALUE "getOrderDetails"));
```

```

call.addParameter("String_1", QNAME_TYPE_STRING, ParameterMode.IN);
String[] params = {orderId};
OrderDetails = (OrderDetails)call.invoke(params);

```

## Stubs and Call Configuration

Developers may want to configure an instance of a stub or a Call interface prior to invoking a service or may prefer to allow the configuration to take place dynamically at runtime. Often, the developer configures the stub or Call interface prior to invoking the service when the service requires basic authentication. For example, a J2SE client application needs to set a user name and password in the stub or Call just before invoking the service; the service requires these two fields so that it can authenticate the client. In other cases, the developer may want flexibility in specifying the endpoint address to use for a particular service, depending on network availability and so forth. The developer might configure this endpoint address dynamically at runtime.

Stubs may be configured statically or dynamically. A stub's static configuration is set from the WSDL file description at the time the stub is generated. Instead of using this static configuration, a client may use methods defined by the javax.xml.rpc.Stub interface to dynamically configure stub properties at runtime. Two methods are of particular interest: `_setProperty` to configure stub properties and `_getProperty` to obtain stub property information. Clients can use these methods to obtain or configure such properties as the service's endpoint address, user name, and password.

Generally, it is advisable to cast vendor-specific stub implementations into a javax.xml.rpc.Stub object for configuration. This ensures that configuration is done in a portable manner and that the application may be run on other JAX-RPC implementations with minimal changes, if any.

Code Example 3.67 shows how a J2EE client might use Stub interface methods to look up and set the endpoint address of a Web service.

### *Example 3.67. Setting Properties on a Stub*

```

Service opcPurchaseOrderSvc =(Service) ic.lookup(AdventureKeys.PO_SERVICE);
PurchaseOrderIntf port = opcPurchaseOrderSvc.getPort(PurchaseOrderIntf.class);
((Stub)port).setProperty(Stub.ENDPOINT_ADDRESS_PROPERTY,
    "http://localhost:8000/webservice/PoEndpointEJB");

```

In a J2EE environment, the J2EE container reserves the right to use two security-related properties for its own purposes. As a result, J2EE clients using any of the three communication modes (stub, dynamic proxy, or DII) should not configure these two security properties:

```

javax.xml.rpc.security.auth.username
javax.xml.rpc.security.auth.password

```

However, J2SE developers do need to set these two security properties if they invoke a Web service that requires basic authentication. When using DII, the client application may set the properties on the Call interface. See Code Example 3.66, which illustrates setting these properties.

J2EE client developers should avoid setting properties other than the javax.xml.rpc.endpoint.address property.

Avoid setting nonstandard properties if it is important to achieve portability among JAX-RPC runtimes. Nonstandard properties are those whose property names are not preceded by javax.xml.rpc.

Avoid using javax.xml.rpc.session.maintain property. This property pertains to a service's ability to support sessions. Unless you have control over the development of both the client and the endpoint, such as when both are developed within the same organization, you cannot be sure that the Web service endpoints support sessions, and you may get into trouble if you set this property incorrectly.

## WSDL-to-Java Type Mapping

When working with Web services, there may be differences between the SOAP-defined types (defined in the WSDL document) used by the service and the Java-defined types used by the client application. To handle these different types, a client of a Web service cannot use the normal approach and import remote classes. Instead, the client must map the WSDL types to Java types to obtain the parameter and return types used by the service. Once the types are mapped, the client has the correct Java types to use in its code.

Generally, the JAX-RPC runtime handles the mapping of parameters, exceptions, and return values to JAX-RPC types. When a client invokes a service, the JAX-RPC runtime maps parameter values to their corresponding SOAP representations and sends an HTTP request containing a SOAP message to the service. When the service responds to the request, the JAX-RPC runtime receives this SOAP response and maps the return values to Java objects or standard types. If an exception occurs, then the runtime maps the WSDL:fault to a Java exception, or to a javax.rmi.RemoteException if a soap:fault is encountered.

The JAX-RPC runtime supports the following standard value types: String, BigInteger, Calender, Date, boolean, byte, short, int, long, float, double, and arrays of these types. Services can return mime types as images mapped to the java.awt.Image class and XML text as javax.xml.transform.Source objects. (The WSDL Basic Profile 1.0 does not support javax.xml.transform.Source objects as mime types. As a result, you should avoid this usage until it is supported by a future WSDL version.) A service may also return complex types, and these are mapped to Java Object representations.

When stubs are used, the JAX-RPC WSDL-to-Java mapping tool maps parameter, exception, and return value types into the generated classes using information contained in the developer-provided WSDL document. Complex types defined within a WSDL document are represented by individual Java classes, as are faults. A WSDL-to-Java mapping tool included with the JAX-RPC runtime also generates classes to serialize and deserialize these values to XML. The JAX-RPC runtime uses these generated classes to serialize parameter values into a SOAP message and deserialize return values and exceptions.

Use WSDL-to-Java tools to generate support classes, even if using the dynamic proxy or DII approach. Whenever possible, developers should try to use a tool to do the WSDL-to-Java mapping, since a tool correctly handles the WSDL format and mapping semantics. Note that the Java objects generated by these mapping tools contain empty constructors and get and set methods for elements. You should use the empty constructor to create an instance of the object and set any field or element values using the corresponding set methods. JAX-RPC does not guarantee that it will correctly map values created as part of the constructor to the corresponding fields.

Although not advisable, it is possible for a developer to work without the benefit of a mapping tool, if none are available. However, without such mapping tools the scope of the developer's work greatly expands. For example, just to compile the client code, the developer must understand the WSDL for a service and generate by hand Java classes that match the parameter and return types defined in the WSDL document or, in the case of a dynamic proxy, the client-side representation of the service endpoint interface. These classes must be set up properly so that the JAX-RPC runtime can match SOAP message types to the corresponding Java objects.

## Processing Return Values

J2EE applications generally use Web components to generate returned data for display. For example, a client accessing an order tracking service might display tracking information in a Web page using an HTML browser. The J2EE component may take the values returned from the Web service and handle them just like any other Java object. For example, a Web component in a J2EE client application might query the status of an order from an order tracking service, which returns these values within a JavaBeans-like object. The client component places the returned object in the request scope and uses a JSP to display its contents. (See Code Example 3.68)

### *Example 3.68. JSP for Generating an HTML Document*

```
<html>
Order ID: ${bean.orderId}
Status: ${bean.status} <br>
Name: ${bean.givenName} ${bean.familyName} <br>
</html>
```

The J2EE platform has a rich set of component technologies for generating Web content, including JavaServer Pages (JSP) technology for generating HTML content and Java Standard Tag Libraries (JSTL). JSTL is a set of tags that assist a client developer in formatting JSPs. For example, JSTL provides additional tags for looping, database access, object access, and XSLT stylesheet transformations. The current version of JSP (2.0), along with future versions, provides an expression language that allows a developer to access bean properties. Together, developers can use JSTL and JSP technologies to generate HTML documents from data retrieved from a service. For example, a developer might generate the following HTML document for the order details (see Code Example 3.69). This HTML document is returned to the HTML browser client that requested the service.

#### **Example 3.69. HTML Document**

```
<html>
    Order: 54321<br>
    Status: SHIPPED<br>
    Name: Duke Smith<br>
</html>
```

A different use case might be EJB components using returned data in a workflow. For example, as the workflow progresses, they may provide order tracking updates by formatting the tracking data in HTML and attaching it to an e-mail message sent to the customer. Unless the initial request originated from a Web tier client, EJB components in a workflow situation do not have Web-tier technologies such as JSP available to them. Furthermore, the order tracking service may return results as XML documents, requiring EJB components to apply XSL transformations to these documents. Code Example DES.E46 shows an XML document containing the returned data from the service.

#### **Example 3.70. Data Returned as XML Document**

```
<orderdetails>
    <id>54321</id>
    <status>SHIPPED</status>
    <shippinginfo>
        <family-name>Smith</family-name>
        <given-name>Duke</given-name>
    </shippinginfo>
</orderdetails>
```

An EJB component may use the XSL stylesheet shown in Code Example 3.71 to transform an order details XML document to the same HTML document as in Code Example 3.69. This HTML document may be attached to an e-mail message and sent to a customer. XSLT transformations may also be used to transform a document into multiple formats for different types of clients.

#### **Example 3.71. XSL Stylesheet**

```
<xsl:stylesheet version='1.0'
    xmlns:xsl='http://www.w3.org/1999/XSL/Transform'>
    <xsl:output method="html"/>
    <xsl:template match="text()" />
    <xsl:template match="orderdetails">
        <html>
            Order: <xsl:value-of select="id/text()"/><br/>
            <xsl:apply-templates/>
        </html>
    </xsl:template>
    <xsl:template match="shippinginfo">
        Name: <xsl:value-of select="given-name/text()"/>
        <xsl:text> </xsl:text>
        <xsl:value-of select="family-name/text()"/>
        <br/>
    </xsl:template>
    <xsl:template match="status">
        Status: <xsl:value-of select="text()"/><br/>
    </xsl:template>
</xsl:stylesheet>
```

Technologies for handling and transforming XML documents are also available to Web components such as servlets or JSPs. Web components may use custom tag components, XSL, or the JAXP APIs to handle XML documents.

Generally, whenever possible client developers should use JSP to generate responses by a service and to present the data as a view to clients (such as browsers that use Web tier technologies).

]For clients in a non-Web tier environment where JSP technology is not available, developers should use XSLT transformations.

Developers can use JSP technology to access content in XML documents and to build an HTML page from the XML document contents. Code Example 3.72 shows how JSP technology makes it easy to parse XML content and build an HTML page from the order detail contents shown in Code Example 3.70.

#### ***Example 3.72. JSP Generating a Web Service Response***

```
<%@ taglib prefix="x" uri="/WEB-INF/x-rt.tld" %>
<x:parse xml="${orderDetailsXml}" var="od" scope="application"/>
<html>
    Order:<x:out select="$od/orderdetails/id"/><br>
    Status:<x:out select="$od/orderdetails/status"/><br>
    Name:<x:out select="$od/orderdetails/shippinginfo/given-name"/>
        <x:out select="$od/orderdetails/shippinginfo/family-name"/>
</html>
```

In this example, the J2EE application first places the order details document received from the service in the request scope using the key orderDetailsXML. The next lines are JSP code that use the x:out JSTL tag to access the order details XML content. These lines of code select fields of interest (such as order identifier, status, and name fields) using XPath expressions, and convert the data to HTML for presentation to a browser client. The JSP code relies on JSTL tags to access these portions of the order details document. For example, the JSTL tag x:out, which uses XPath expressions, accesses the order identifier, status, and name fields in the order details document. When the JSP processing completes, the result is identical to the HTML page shown in Code Example DES.E45.

## **Packaging**

To access a service, a stand-alone client requires a runtime environment. For J2SE clients, the runtime must be packaged with the application. J2EE clients rely on the JAX-RPC runtime.

J2ME clients do not need to package the JAX-RPC runtime with the applications. Although stubs do need to be packaged with an application, the stubs are portable across JAX-RPC runtimes. The portability of stubs is critical because J2ME clients cannot generate or compile stub implementation code, and thus must rely on more dynamic provisioning.

## ***J2EE Clients***

Web service clients running in a J2EE environment require some basic artifacts, as follows:

- Service reference— A service-ref element in the deployment descriptor
- Mapping file— A JAX-RPC mapping file
- WSDL document
- Service endpoint interface— A stub or dynamic proxy
- Generated classes

The service-ref element, part of the general J2EE 1.4 schema, contains information about a service. Web, EJB, and J2EE application client module deployment descriptors use this element to locate the JAX-RPC mapping files as well as the service's WSDL file. The service reference element maps a service to a JNDI resource name and also specifies the service endpoint interface for those clients using stubs and dynamic proxies. (Clients using DII do not need to specify the service endpoint interface.) It also specifies the WSDL file for the service (its location is given relative to the root of the package) and the qualified name for the service in the WSDL file. If a WSDL

file is required, the element specifies a JAX-RPC mapping file. The mapping file's location is also relative to the package root. Code Example 3.73 is an example of a service reference:

#### **Example 3.73. web.xml Fragment for Web Service Reference**

```
<service-ref>
    <description>OPC OT Service Client</description>
    <service-ref-name>service/OpcOrderTrackingService
    </service-ref-name>
    <service-interface>
        com.sun.j2ee.blueprints.adventure.web.actions.OpcOrderTrackingService
    </service-interface>
    <wsdl-file>WEB-INF/wsdl/OpcOrderTrackingService.wsdl
    </wsdl-file>
    <jaxrpc-mapping-file>WEB-INF/opc-ot-jaxrpc-mapping.xml
    </jaxrpc-mapping-file>
    <service-qname
        xmlns:servicens="urn:OpcOrderTrackingService">
        servicens:OpcOrderTrackingService
    </service-qname>
</service-ref>
```

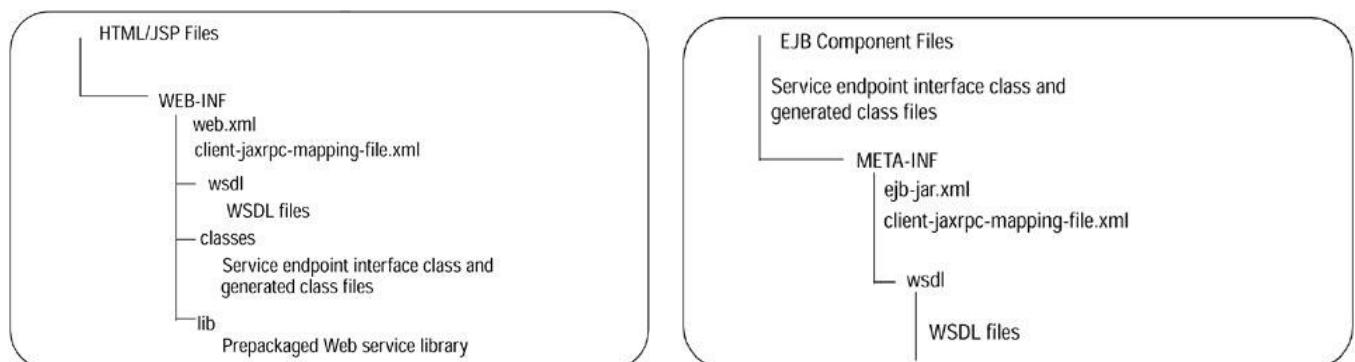
The JAX-RPC mapping file specifies the package name containing the generated runtime classes and defines the namespace URI for the service. (See Code Example 3.74)

#### **Example 3.74. JAX-RPC Mapping File**

```
<java-wsdl-mapping xmlns="http://java.sun.com/xml/ns/j2ee"
    xmlns:xsi="http://www.w3.org/2001/XMLSchema-instance"
    xsi:schemaLocation="http://java.sun.com/xml/ns/j2ee
    http://www.ibm.com/webservices/xsd/j2ee_jaxrpc_mapping_1_1.xsd"
    version="1.1">
    <package-mapping>
        <package-type>com.sun.j2ee.blueprints.adventure.web.actions
    </package-type>
    <namespaceURI>urn:OpcOrderTrackingService</namespaceURI>
    </package-mapping>
</java-wsdl-mapping>
```

WSDL files, including partial WSDL files, are packaged within clients. Their location is dependent on the type of module. Since clients using DII do not require a WSDL file, they leave the wsdl-file element portion of the service-ref element undefined and they must not specify the jaxrpc-mapping-file element.

For a web application archive (WAR) file, the WSDL file is in the WEB-INF/wsdl directory. (See Figure 3.46(a)) For an EJB endpoint as well as a J2EE application client, the WSDL file is in the directory MET-INF/wsdl. (See Figure 3.46(b)) Both directories are relative to the root directory of the application module.



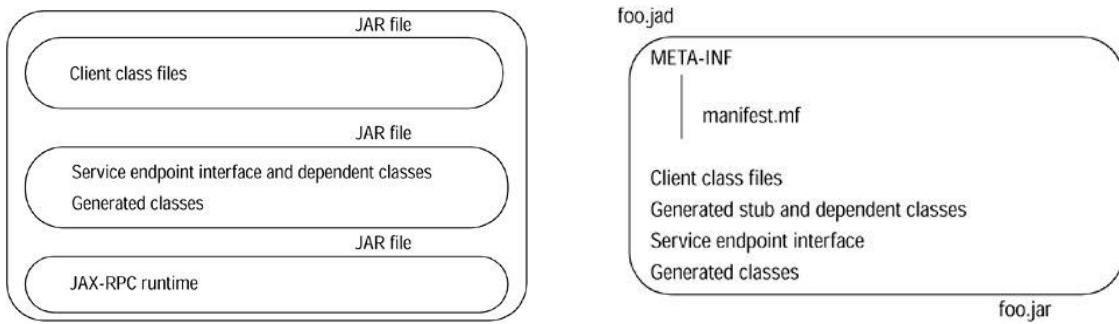
**Fig. 3.46. (a) Web Application Module Packaging (b) EJB Module Packaging**

For Web tier clients, a service-ref element in the web.xml file contains the location of the JAX-RPC mapping file, client-jaxrpc-mapping-file.xml. The service endpoint interface (if provided) is either a class file in the WEB-INF/classes directory or it is packaged in a JAR file in the WEB-INF/lib directory. Generated classes are located in the same directory.

For EJB tier client components, the service-ref element is defined in the deployment descriptor of the ejb-jar.xml file and the client-jaxrpc-mapping-file.xml mapping file. The WSDL files are in a META-INF/wsdl directory. The service endpoint interface as well as generated class files are stored in the module's root directory.

The client developer should ensure that the resource-ref definition in the client deployment descriptor is correct and that the JAX-RPC mapping file is packaged in the correct module.

**J2SE clients** using stubs or dynamic proxies should package the service endpoint interface with the application client, and they should be referenced by the class path attribute of the package's manifest file. J2SE clients also must provide a JAX-RPC runtime. For example, a J2SE client is packaged along with its supporting classes or with references to these classes. (See Figure 3.47) The service endpoint interface as well as the necessary generated files may be provided in a separate JAR file.



**Fig. 3.47. (a) Packaging a J2SE Client with Web Service Library (b) Packaging a MIDlet Web Service Client Application**

Figure 3.47(a) shows how to package a J2SE client in a modular manner. The classes specific for Web service access are kept in a separate JAR file referenced via a class path dependency. Packaged this way, a client can swap out the service endpoint access without having to change the core client code. The service access classes may also be shared by different application clients accessing the same service. A developer utilizing a prepackaged service interface may also be able to develop a Web service client with less knowledge of a service.

**J2ME Clients.** Two optional packages, both of which are extensions of the J2ME platform, enable Web services in the J2ME platform by providing runtime support for XML processing and JAX-RPC communication. (Note that although XML processing capabilities are not provided in the J2ME platform, they are required for JAX-RPC communication.)

A J2ME client application developer must package certain resources with a J2ME application. First, J2ME applications are packaged in a MIDlet format. A MIDlet is a Java Archive (JAR) file that contains class files, application resources, and a manifest file (manifest.mf), which contains application attributes.

A developer may also provide an external Java Application Descriptor (JAD) file for the MIDlet. A JAD file provides the J2ME environment with additional information about the application, such as the location of the application's MIDlet file. A JAD file's attributes mirror those found in the manifest file, but the JAD file takes precedence over the manifest file. Furthermore, a developer or deployer may override application attributes in the JAD file. Figure 3.47(b) describes the packaging for a Web service MIDlet client application.

The foo.jar MIDlet file contains the client application classes and the respective artifacts generated by the J2ME Web service development tools, as well as a manifest file. A foo.jad file describes the foo.jar MIDlet. Similar to the J2EE platform, the J2ME platform with the optional Web service packages provides the resources required for Web service communication.

### 3.3.10.16 Java Web Services Developer Pack (JWSDP)

Sun Microsystems as part of its Java community process has released its Java API for Web Services for the developer community as the Java Web Services Developer Pack (JWSDP). It provides a full-fledged solution package for developing and testing Web services using the Java APIs.

JWSDP provides a one-stop Java API solution for building Web services using a Java platform. The key API components include the following:

- Java API for XML Messaging (JAXM)
- Java API for XML Processing (JAXP)
- Java API for XML Registries (JAXR)
- Java API for XML Binding (JAXB)
- Java API for XML-Based RPC (JAX-RPC)
- Java WSDP Registry Server (JWSDP)
- Java Server Pages Standard Tag Library (JSTL)

### 3.3.10.17 Other Java-XML Technologies

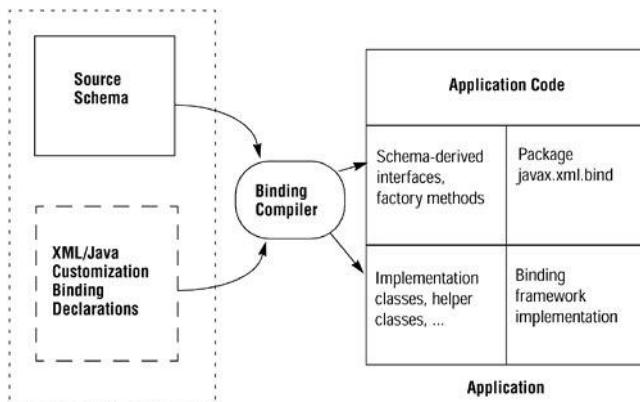
Up to now, we have discussed the Web service-specific technologies that are a mandatory part of the J2EE platform. As such, these technologies must be present in any J2EE implementation from any vendor. Apart from these, there are other Java-XML technologies that, while not a mandatory requirement of the J2EE platform, still prove very useful for implementing Web services. While there are a number of such technologies, we discuss here only those referenced throughout this book. One such non-mandatory but useful Java-XML technology is the Java Architecture for XML Binding (JAXB), which standardizes the representation of an XML document as an in-memory object.

As we have already seen, when two parties communicate by passing XML documents between them, the XML documents should follow some structure so that the communicating parties can understand the contents of the documents. XML document structure is defined using the standard schema facility for XML documents. Of course, while developers can use a DOM or SAX parser to parse such documents, it is much easier if the various parts of the XML documents are mapped or bound to in-memory objects that truly represent the document's intended meaning, as per the schema definition. In addition to using these objects, developers have access to the schema definitions as part of their logic. Such a facility is commonly called an XML data-binding facility. JAXB provides a good quality XML data-binding facility for the J2EE platform. Figure 3.48 shows the overall architecture of the JAXB data-binding facility.

Table 3.16 summarizes the standards supported by the different J2EE platform technologies.

**Table 3.16. J2EE Platform Web Service Support**

Technology Name	Supporting Standard	Purpose
JAXP	XML schema	Enables processing of XML documents in a vendor neutral way; supports SAX and DOM models
JAX-RPC	SOAP	Enables exchange of SOAP requests and responses through an API that hides the complex SOAP details from the developers
JAXR	UDDI, ebXML	Enables accessing business registries with an API that supports any type of registry specification
SAAJ	SOAP with Attachments	Enables exchange of document-oriented XML messages using Java APIs
J2EE for Web Services	Integrates Java XML technologies into the J2EE platform; supports WS-I Basic Profile	Enables development and deployment of portable and interoperable Web services on the J2EE platform
JAXB (optional)	Standard in-memory representation of an XML document	Provides an XML data-binding facility for the J2EE platform



*Figure 3.48. JAXB Architecture*

### 3.3.10.18 JAXB and XML Over HTTP Protocol

JAXB consists of three main components:

- A binding compiler that creates Java classes (also called content classes) from a given schema. Complex type definitions within the schema are mapped to separate content classes, while simple types (such as attribute/element declarations) are mapped to fields within a content class. Developers use get and set methods (similar to JavaBeans get and set methods) to access and modify the object contents.
- A binding framework that provides runtime services—such as marshalling, unmarshalling, and validation—that can be performed on the contents classes.
- A binding language that describes binding of the schema to Java classes. This language enables a developer to override the default binding rules, thereby helping the developer to customize the content classes that are created by the binding compiler.

Apart from JAXB, there are other Java technologies that support Web service standards in terms of long-lived transactions, business process workflow, and so forth. At the time of this writing, they have not been finalized and hence will be dealt with in a future version of this book.

The process of designing services yields to a number of results. First of all, it produces a list of service definitions which is sometime referred to as a "Catalog of Services". This, naturally, should not have the form of a flat list, but will be organized into sections or "Functional Domains". So we could have, for example, the "Item", "Order", and "Customer" functional domains. Under these functional domains, the following services can be defined:

Item Functional domains are as follows:

```
insertItem
updateItem
deleteItemById
findItemById
findAllItems
findItemsByCriteria
```

Order Functional domains are as follows:

```
createOrder
findOrderById
findAllOrdersByCustomer
```

Often there will be the need for some "orthogonal" services. In fact, some services could share common mechanisms, such as a control flow or transaction handling.

Generally, at the beginning of any software design process, a common task is to focus on the basic domain objects and their essential handling operations: **Create** (or insert), **Read** (select), **Update**, and **Delete**. This is usually referred to as **CRUD**. No matter which language you are using, which architecture is adopted, all projects virtually have to deal with CRUD actions. In our case, a basic domain object is the Item entity and the first four listed services are exactly the CRUD functions.

Let's begin by analyzing and designing the services that handle the item domain. The insertItem service, for example, could have the following form. The client who wants to use this service in order to insert a new item must provide an XML message with the above input schema. Note that the item id has a zero value, since it is assumed that the server will assign it and return to the client along with a return code.

Input	Service	Output
<pre>&lt;Item&gt;   &lt;id&gt;0&lt;/id&gt;   &lt;code&gt;RX004&lt;/code&gt;   &lt;description&gt;     Eth. Cable   &lt;/description&gt; &lt;/Item&gt;</pre>	=> insertItem =>	<pre>&lt;Result&gt;   &lt;retCode&gt;     OK   &lt;/retCode&gt;   &lt;id&gt;137&lt;/id&gt; &lt;/Result&gt;</pre>
<pre>&lt;ItemId&gt;   &lt;id&gt;137&lt;/id&gt; &lt;/ItemId&gt;</pre>	=> findItemById =>	<pre>&lt;Item&gt;   &lt;id&gt;137&lt;/id&gt;   &lt;code&gt;     RX004   &lt;/code&gt;   &lt;description&gt;     Eth. Cable 4 ft.   &lt;/description&gt; &lt;/Item&gt;</pre>
<pre>&lt;Item&gt;   &lt;id&gt;137&lt;/id&gt;   &lt;code&gt;RX004&lt;/code&gt;   &lt;description&gt;     Eth. Cable 4 ft.   &lt;/description&gt; &lt;/Item&gt;</pre>	=> updateItem =>	<pre>&lt;Result&gt;   &lt;retCode&gt;     OK   &lt;/retCode&gt;   &lt;id&gt;137&lt;/id&gt; &lt;/Result&gt;</pre>
<pre>&lt;ItemId&gt;   &lt;id&gt;137&lt;/id&gt; &lt;/ItemId&gt;</pre>	=> deleteItem =>	<pre>&lt;Result&gt;   &lt;retCode&gt;     OK   &lt;/retCode&gt;   &lt;id&gt;0&lt;/id&gt; &lt;/Result&gt;</pre>

The above design is just an example of the possible communication protocol we can adopt. In fact, in this scenario, without constraints or patterns to follow, we are free to decide the communication protocol. For example, we could find it better to have a unique input-output pattern and a single entry point for all CRUD methods. Here the input is a message with the service name embedded, along with the item object, while the output is composed by a return code and an item object, as described in the following figure.

### Generic CRUD Action

Input	Service	Output
<pre>&lt;ItemAction&gt;   &lt;method&gt;     findById   &lt;/method&gt;   &lt;item&gt;     &lt;id&gt;137&lt;/id&gt;     &lt;code&gt;&lt;/code&gt;     &lt;description&gt;     &lt;/description&gt;   &lt;/item&gt; &lt;/ItemAction&gt;</pre>	=> itemCrudService =>	<pre>&lt;ItemActionResponse&gt;   &lt;retCode&gt;OK&lt;/retCode&gt;   &lt;item&gt;     &lt;id&gt;137&lt;/id&gt;     &lt;code&gt;RX004&lt;/code&gt;     &lt;description&gt;       Eth. Cable 4 ft.     &lt;/description&gt;   &lt;/item&gt; &lt;/ItemActionResponse&gt;</pre>

Here the advantage of having a single service for all CRUD actions has a price: we have to provide a partially

filled item object (with just the id attribute valued) while invoking the service with findById and delete methods (in fact only the insert and the update methods need to really pass the full-valued item object). On the other hand, having a filled item into the response is meaningful just for the findById method.

However, the CRUD actions do not cover, generally, all the needed services. For instance, in the Item domain, we also need a method that retrieves all the items or at least a subset of them. A possible specification of this service could be the following:

### Non-CRUD Action

Input	Service	Output
void input	=> findAllItems =>	<pre>&lt;Items&gt;   &lt;item&gt;     &lt;id&gt;137&lt;/id&gt;     &lt;code&gt;RX004&lt;/code&gt;     &lt;description&gt;       Eth. Cable 4 ft.     &lt;/description&gt;   &lt;/item&gt;   ... &lt;/Items&gt;</pre>

So far, we have explored a couple of feasible paths that the service designer may follow. As you noticed, the communication protocol is completely up to you. There are no guidelines, just your skill to abstract concepts.

Once we have settled for the communication protocol (may be one of the above or yet another of your choice), we need to think about the layer protocol and its details. The HTTP protocol is a very practical and flexible solution: we can send the XML message as an HTTP request. This approach is also known as POX-over-HTTP, where **POX** stands for **Plain Old Xml**.

In practice, we just need an XML translation library in order to transform the objects written in our programming language into XML documents and vice versa. This is all we need for implementing the services we have described above. But there is more. We can even use different languages for implementing the client and the server side as long as each layer adheres to the defined protocol. The XML document is the key to decoupling parts, as shown in the Figure 3.49, where a possible scenario is depicted.

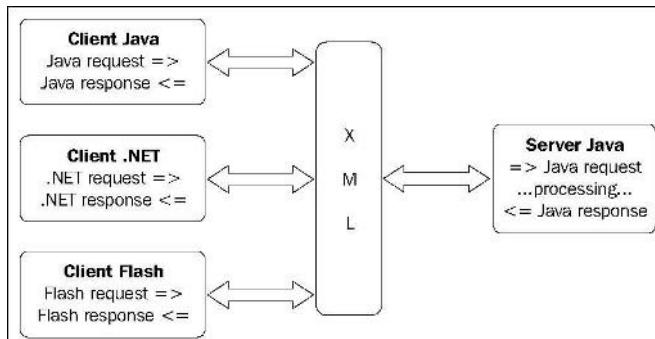


Fig. 3.49 Web Service Decoupling

### A Basic Java Implementation of POX-over-HTTP

Now, to complete the example, we will see how to implement the findById service in the Java language. For the automatic binding, we decided to use the **JAXB library**. This component is included into the latest **Java 6 JDK**. So if you use this version of Java you do not need any additional jars. Otherwise, you will have to download JAXB and add it explicitly to your server and client class paths. However, since we will make use of Java Annotations, the following source code requires at least a JDK 5 release in order to be compiled and executed. For the server side service implementation, we adopted Tomcat 5.5. The implementation we are stepping through is in fact made up of a simple Java Servlet.

Let's start with the classes that will be exchanged between the client and the server: Item, ItemAction, and ItemActionResponse. There is very little indeed to say about them; they are basic **POJO (Plain Old Java Object)** with a Java annotation that has a key role in the process of XML serialization/deserialization.

### **Example 3.75. JXB.EI—XML Binding Annotations**

```
@XmlRootElement(name="Item")
public class Item {
    private int id;
    private String code;
    private String description;
    ... getter/setter methods omitted ...
@XmlRootElement(name="ItemAction")
public class ItemAction{
    private String method;
    private Item item;
    ...
@XmlRootElement(name="ItemActionResponse")
public class ItemActionResponse {
    private String retCode
    private Item item;
    ...
}
```

The following code represents our main goal: the service implementation. It is implemented in the doPost() method of a Servlet that was mapped with the url-pattern /itemCrudService in the application descriptor web.xml.

### **Example 3.76 —ItemCrudService Server Implementation**

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
                      throws ServletException, IOException
{
    try{
        JAXBContext jaxbContext = JAXBContext.newInstance
            (ItemAction.class, ItemActionResponse.class);
        Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
        //Receiving the XML request and transform it into a Java object
        ItemAction itemAction = (ItemAction)
            unmarshaller.unmarshal(request.getInputStream());
        //Do some action depending on the request content
        String method = itemAction.getMethod();
        //Prepare the response as a Java object
        ItemActionResponse itemActionResponse = new ItemActionResponse();
        if ("findById".equals(method)) {
            int id = itemAction.getItemId();
            //Retrieve item (e.g. from db)
            Item item = new Item();
            item.setId(id);
            item.setCode("Item XYZ");
            item.setDescription("Description item XYZ");
            //Fill the response
            itemActionResponse.setRetCode("OK");
            itemActionResponse.setItem(item);
        }
        Marshaller marshaller = jaxbContext.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
        //The following line is not required, it was inserted
        //just to see the content of the generated XML message
        marshaller.marshal(itemActionResponse, System.out);
        //Send the XML message to the client
        marshaller.marshal( itemActionResponse, response.getOutputStream());
    }
    catch (JAXBException e){
        throw new ServletException(e);
    }
}
```

We have just written a basic web service; the flow is clear:

1. Deserialize the XML request

2. Do the processing
3. Prepare and serialize the response

Please note that the above service can be invoked by any language or technology, as long as the process of XML serialization or deserialization is available and the communication protocol is known on the client side.

The Java code for testing the service is as follows:

**Example 3.77—ItemCrudService Client Request**

```
//Prepare the request
ItemAction itemAction = new ItemAction();
Item item = new Item();
item.setId(26);
itemAction.setMethod("findById");
itemAction.setItem(item);
//Prepare and establish the connection with the service
URL url = new URL("http://localhost/SoaBookPoxHttp/itemCrudService");
HttpURLConnection con = (HttpURLConnection) url.openConnection();
con.setDoOutput(true);
//Set the HTTP request method
con.setRequestMethod("POST");
con.connect();
JAXBContext jaxbContext = JAXBContext.newInstance
    (ItemAction.class, ItemActionResponse.class);
Marshaller marshaller = jaxbContext.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);

//The following line is not required, it was inserted
//just to see the content of the generated XML message
marshaller.marshal(itemAction, System.out);
//Send the XML request to the service
marshaller.marshal(itemAction, con.getOutputStream());

//Get the XML response from the service and deserialize it
Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
ItemActionResponse itemActionResponse = (ItemActionResponse)
    unmarshaller.unmarshal(con.getInputStream());
//Show the response content
System.out.println("retCode="+itemActionResponse.getRetCode() + "\r" +
    "id=" + itemActionResponse.getItem().getId() + "\r" +
    "code=" + itemActionResponse.getItem().getCode() +
    "\r" + "description=" + itemActionResponse.getItem() +
    .getDescription());
```

As you see, on the client side, it is mandatory to have the visibility of all the classes involved in the communication process (Item, ItemAction, and ItemActionResponse). In this case, where Java is used both for service implementation and client development, these classes were just copied from the server side and dropped into the client project. In general, of course, this is not a requirement (think about using different languages). The only requirement is having objects that fit the serialization or deserialization process.

Using the above approach, in order to implement the findAllItems service, we should create another servlet that does not need to retrieve any input and returns a list of items:

**Example 3.78—findAllItems Service Implementation**

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
    throws ServletException, IOException
{
    try {
        JAXBContext jaxbContext = JAXBContext.newInstance
            (ItemList.class, Item.class);
        ItemList itemList = new ItemList();
        itemList.setList(new ArrayList());
```

```

Item il = new Item();
// il.set ... ;
itemList.getList().add(il);

// ... populate itemList ...
Marshaller marshaller = jaxbContext.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);

//Just to see the content of the generated XML message
marshaller.marshal(itemList, System.out);
//Send the XML message to the client
marshaller.marshal(itemList, response.getOutputStream());
}

catch (JAXBException e) {
    throw new ServletException(e);
}
}
}

```

Note that we also need to define the `ItemList` class:

#### **Example 3.79—*ItemList Binding***

```

import java.util.List;
import javax.xml.bind.annotation.XmlRootElement;

@XmlRootElement(name="ItemList")
public class ItemList {
    private List list;
    ...

```

While the correspondent client code may look like this:

#### **Example 3.80—*findAllItems Service Client Request***

```

URL url = new URL("http://localhost/SoaBookPoxHttp/findAllItems");
HttpURLConnection con = (HttpURLConnection) url.openConnection();
con.setRequestMethod("POST");
con.connect();

//Void Request
//Get Response
 JAXBContext jaxbContext = JAXBContext.newInstance(ItemList.class, Item.class);
Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
ItemList itemList = (ItemList) unmarshaller.unmarshal(con.getInputStream());
for (Iterator iterator = itemList.getList().iterator(); iterator.hasNext();) {
    Item item = (Item) iterator.next();
    System.out.println( item.getId() + " - " + item.getCode() + " - " +
        item.getDescription());
}

```

The **Representational State Transfer (REST)** is a web architectural style. The basic idea of REST is the full exploitation of the HTTP protocol, in particular:

- It focuses on **Resources**, that is, each service should be designed as an action on a resource.
- It takes full advantage of all **HTTP verbs** (not just GET and POST, but also PUT and DELETE).

In the basic POX-over-HTTP example, which we saw previously, you may have noticed that we assumed to use POST as HTTP verb. Although, we listed the source code for just one out of the several services we defined, all of them can be implemented using the same verb (not necessarily POST, any other method will do the job right). Therefore, the idea is, why not exploit this transportation protocol feature to map the usual CRUD and other methods in order to have a clearer communication protocol?

An association between the four CRUD methods and the correspondent HTTP verbs was therefore established and is shown in Table 3.17:

**Table 3.17 Association between CRUD methods and HTTP verbs**

HTTP verb	CRUD action	Action description
POST	CREATE	Save new resources
GET	READ	Read resources
PUT	UPDATE	Modify existing resources
DELETE	DELETE	Delete resources

Keeping in mind that the set of resources, each one with its values, represents the State of the system, the following rules should be applied:

- The State can be modified by verbs POST, PUT, and DELETE.
- The State should never change as a consequence of a GET verb.
- The verb POST should be used to **add** resources to the State.
- The verb PUT should be used to **alter** resources into the State.
- The verb DELETE should be used to **remove** resources from the State.
- The communication protocol should be **stateless**, that is, a call should not depend on the previous ones.

All this may sound quite interesting, but what exactly is a "resource"? Basically, a resource is a scope within which the four HTTP verbs can cover all requested actions. Take for instance the Item domain we explored earlier. Well, that is a good candidate to be treated as a resource. As you see in the implementation source (Listing 5), a switch was introduced (in the if-then block) to do different actions depending on the method name contained in the request. This conditional flow control can be avoided and its role can be done by the HTTP verbs. Let's see how to do that in practice:

In order to create a new Item (the insertItem service), an XML document containing the object can be sent to the server with the POST method, while the response may be a generic outcome object:

#### **Example 3.81—Outcome Binding**

```
@XmlElement(name="Outcome")
public class Outcome {
    private String retCode;
    private String retMessage;
    ...
}
```

The service code will then be kept small and without the conditional flow instructions for the action to be executed:

#### **Example 3.82—REST CREATE Server Implementation**

```
protected void doPost(HttpServletRequest request, HttpServletResponse response)
                     throws ServletException, IOException
{
    try{
        JAXBContext jaxbContext = JAXBContext.newInstance
                               (Item.class, Outcome.class);
        Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
        //Receiving the XML request and transform it into a Java object
        Item item = (Item) unmarshaller.unmarshal(request.getInputStream());
        System.out.println("Inserting item# "+item.getId());
        // ... insert item
        Marshaller marshaller = jaxbContext.createMarshaller();
        marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
        Outcome outcome = new Outcome();
        outcome.setRetCode("OK");
        outcome.setRetMessage("Item was inserted successfully");
        marshaller.marshal(outcome, response.getOutputStream());
    }
}
```

```

        catch (Exception e) {
            throw new ServletException(e);
        }
    }
}

```

Please note that the above is a simple homemade implementation of the REST protocol. We decided to use a basic servlet implementation in order to focus on the key concepts of this communication protocol. However, there are several other ways to adopt REST, for example, by using JAX-WS (which will be used next when exploring SOAP) or with Axis 2.

Coming back to our example, the updateItem service can be implemented by analogous source code with the only differences being the servlet method (doPut instead of doPost) and, of course, the inner update action. Indeed, the REST approach would recommend another difference that we will show a little ahead.

For the insert action, here is an example of the client code:

**Example 3.83—REST CREATE Client Request**

```

Item item = new Item();
item.set...
//Prepare and establish the connection with the service
URL url = new URL("http://localhost/SoaBookREST/itemService");
HttpURLConnection con = (HttpURLConnection) url.openConnection();
con.setDoOutput(true);
//Set the HTTP request method
con.setRequestMethod("POST");
con.connect();
JAXBContext jaxbContext = JAXBContext.newInstance(Item.class, Outcome.class);
Marshaller marshaller = jaxbContext.createMarshaller();
marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);

//Send the XML request to the service
marshaller.marshal(item, con.getOutputStream());

//Get the XML response from the service and deserialize it
Unmarshaller unmarshaller = jaxbContext.createUnmarshaller();
Outcome outcome = (Outcome)
unmarshaller.unmarshal(con.getInputStream());

```

As far as the deleteItem, findItemById, and findAllItems services are concerned, the REST methodology suggests a slightly different communication protocol. In fact, the above actions do not need to upload an XML document. They just have to pass the object's id or nothing at all (in the findAllItems case). In these situations, the called URI, along with the HTTP verb, contains all the information needed to perform the actions. The samples of the REST requests are as shown in Table 3.18.

**Table 3.18. Verb and actions**

Verb	URI sample	Action
DELETE	http://localhost/SoaBookREST/itemService/14	Delete item #14
GET	http://localhost/SoaBookREST/itemService/14	Retrieve item #14
GET	http://localhost/SoaBookREST/itemService	Retrieve all items

As you can see, the HTTP request (verb plus URI) tells clearly what is happening, or at least, what is the desired action. Note that now the URI may contain some additional data (the id). So in the web descriptor, we must check to have an URL pattern instead of an exact correspondence:

**Example 3.84—servlet Mapping Section in web.xml**

```

<servlet-mapping>
    <servlet-name>ItemService</servlet-name>
    <url-pattern>/itemService/*</url-pattern>
</servlet-mapping>

```

Here is the service code for the last two actions:

#### **Example 3.85—REST READ Service Implementation**

```
protected void doGet(HttpServletRequest request, HttpServletResponse response)
                     throws ServletException, IOException
{
    try{
        if (request.getPathInfo()==null){
            //findAllItems
            ItemList itemList = new ItemList();
            itemList.setList(new ArrayList());
            //retrieve all items
            ...
            itemList.getList().add(...);
            ...
            //Send the XML message to the client
            JAXBContext jaxbContext = JAXBContext.newInstance
                (ItemList.class, Item.class);
            Marshaller marshaller = jaxbContext.createMarshaller();
            marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
            marshaller.marshal(itemList, response.getOutputStream());
        } else {
            //findItemById
            int id = (new Integer(request.getPathInfo().substring(1)))
                .intValue();
            //retrieve item by id (e.g. from a database)
            Item item = ...
            JAXBContext jaxbContext = JAXBContext.newInstance(Item.class);
            Marshaller marshaller = jaxbContext.createMarshaller();
            marshaller.setProperty(Marshaller.JAXB_FORMATTED_OUTPUT, Boolean.TRUE);
            marshaller.marshal(item, response.getOutputStream());
        }
    }
    catch (Exception e) {
        throw new ServletException(e);
    }
}
```

As far as the **update** action is concerned, the REST style indeed suggests having the object's id in the URI:

PUT http://localhost/SoaBookREST/itemService/14

In fact, this way, the HTTP request is self-explanatory (it reads "update item #14") in accordance with the REST philosophy.

In general, if you need to create other non-CRUD services with the above technology, the choice falls between:

1. Creating an ad-hoc servlet to be used with an appropriate HTTP verb
2. Re-using an existing servlet and verb with the introduction of some custom logic in the request composition and parsing

Also consider that passing parameters into the request can be a practical alternative to differentiate the control flow, although REST-purists do not like this approach. So, the request could take the form:

http://localhost/SoaBookREST/itemService?id=14

REST purists may have a point here. In fact, this way we are introducing a dependency on the parameter name (id), and somehow adding complexity to a simple and linear style.

### 3.3.10.19 JAX-WS 2

It is a specification (JSR-224), rather than an implementation. In fact, its name stands for "**Java API for XML Web Services**" and it follows the previous **JAX-RPC (Java API for XML-based Remote Procedure Call)** specification. The JAX-WS 2.0 specification replaces JAX-RPC 1.0 and is the next generation web services API-based on JSR 224. One of the main advantages of this choice is that its Reference Implementation is included both into Java SE 6 and Java EE 5. Thus no external library is needed in order to use it.

As far as the data binding model is concerned, it uses JAXB, while the XML parser engine is the stream-based pull parser **StAX (Streaming API for XML)**. This parsing approach, where the client gets XML data only when requested, allows better performances compared to a DOM-based approach, where the entire XML document is parsed to obtain an in-memory object tree. In fact, not only is the requested memory footprint smaller, but also there is the advantage of having the parser start its process earlier.

JAX-WS supports both SOAP and REST communication protocols, though RESTful services cannot take advantage of the automatic code generation (in contrast with the SOAP/WSDL approach) since a standard for RESTful services description has not yet been defined.

It strongly relies on the usage of annotations and supports a number of transportation protocols, going beyond HTTP including SMTP and JMS. Moreover the range of WS-\* features covered is indeed wide. It goes from WS-Security and Policy to WS-Atomic Transaction and WS-ReliableMessaging.

Finally, among its features, there is also the capability to build Stateful web services.

The JAX-WS 2.0 project develops and evolves the code base for the reference implementation of the JAX-WS specification and is available in the URL <https://jax-ws.dev.java.net/>. At present the code base supports JAX-WS 2.0 and JAXWS 2.1.

The following list specifies new features implemented by JAX-WS 2.0:

- Direct support for JAXB 2.0-based data binding
- Support for the latest W3C and WS-I standards (e.g. SOAP 1.2, WSDL 1.2, and SAAJ 1.3)
- Standardized metadata for Java to WSDL (and vice versa) mapping
- Ease-of-development features
- Support for easier evolution of web services
- Improved handler framework
- Support for asynchronous RPC and non-HTTP transports

Another exciting feature of JAX-WS 2.0 is the support it has got within Java Platform, Standard Edition 6 (Java SE 6). This means, JAX-WS 2.0-based code and components can be executed from within a fully blown J2EE server infrastructure such as Project GlassFish, or with just Java SE 6. This is indeed a great advantage for Java developers, which has been previously enjoyed only by .NET developers (.NET stack supports web services development in the light-weight manner).

JAX-WS 2.0 provides the following new APIs in the Java SE 6 platform to build web applications and web services:

API	Package
JAX-WS	javax.xml.ws
SAAJ	javax.xml.soap
WS Metadata	javax.jws

**Using JAX-WS for SOAP.** In order to show you a straightforward implementation of the services of our domain sample, we will use the JDK 6 embedded capabilities to define and publish web services. In fact, while in the previous examples we exploited the JAXB component to perform the automatic binding between XML documents and Java objects, here we will make use of the JAX-WS library, which will increase the abstraction level significantly.

Look how easy it is to create a couple of services such as insert and update:

#### Example 3.86—JAX-WS Annotations

```
package com.packt.soajava.soap.service.item;

import javax.jws.WebMethod;
import javax.jws.WebService;
import javax.xml.ws.Endpoint;
import com.packt.soajava.model.item.Item;
import com.packt.soajava.model.item.Outcome;

@WebService
public class ItemWs {

    @WebMethod
    public Outcome insert(Item item) {
        //Insert item ...
        System.out.println("Inserting item "+item.getId());
        Outcome outcome = new Outcome();
        outcome.setRetCode("OK");
        outcome.setRetMessage("Item was inserted successfully");
        return outcome;
    }

    @WebMethod
    public Outcome update(Item item) {
        //Update item ...
        System.out.println("Updating item "+item.getId());
        Outcome outcome = new Outcome();
        outcome.setRetCode("OK");
        outcome.setRetMessage("Item was updated successfully");
        return outcome;
    }
}
```

As you see the abstraction level here allows a compact and neat source code, without any part dealing with the serialization/deserialization process, just the essential service code. The structure of a web service implemented with SOAP is quite different from what we have seen with the other approaches so far. In fact with basic POX-over-HTTP, we had to create several classes (servlets) for a single functional domain (for example, Item domain), while using REST a single class was needed, but only because our designed methods matched well with the four HTTP verbs. With SOAP, we can have as many methods in a web service as we need, and each one is independent from the other in its signature.

Publishing the above service requires some further steps. First of all, we need to generate the classes involved in the communication process. This is performed by the wsgen utility bundled with JDK 6. Just open a command shell, and run the following line:

```
<JDK6_HOME>\bin\wsgen -cp <ProjectClassesRoot> -d <ProjectSourceRoot> -keep
com.packt.soajava.soap.service.item.ItemWs
```

The above command will generate the needed classes into the <ProjectSourceRoot> (-d=destination directory) folder, given the specified full-path ItemWs class and the classpath (-cp). Now, you should find a new package in the project sources (com.packt.soajava.soap.service.item.jaxws), and inside it, there should be four classes. In fact, for each defined web method, two classes will be generated: one with the same name of the method (capitalized) and the other with the same name concatenated with "Response".

The web service is now ready to be published. The JDK 6 makes available, mainly for prototyping usage, a very easy way to do this. Just write and run a class that executes the following line:

```
Endpoint.publish( "http://localhost:8001/SoaBookSOAP_server/itemWs", new ItemWs());
```

We have just published our web service at the specified URI. You may of course change this URI in order to change the URL pattern or port.

How can we check if the service was published correctly? Point your browser to the correspondent WSDL:

```
http://localhost:8001/SoaBookSOAP_server/itemWs?WSDL
```

What you are looking at is the automatically generated WSDL. Its content represents the structure of the service, and it plays a key role when it comes to have the client classes automatically generated.

The latter action can, in fact, be performed by the wsimport utility:

```
<JDK6_HOME>\bin\wsimport -d <ClientProjectSourceRoot>
-p com.packt.soajava.soap.client.item
-keep http://localhost:8001/SoaBookSOAP_server/itemWs?WSDL
```

With this command the needed classes will be created into the specified client source folder (-d), using the given package name (-p), and retrieving the service structure at the given URI. Among these classes we will find ItemWsService, the client factory of the web service and ItemWs, an interface supported by the service proxy created by the ItemWsService factory.

Now the client code can be as simple as this:

**Example 3.87—JAX-WS Sample Client**

```
ItemWsService service = new ItemWsService();
ItemWs itemWs = service.getItemWsPort();

Item item1 = new Item();
item1.set ...
Outcome outcome = itemWs.insert(item1);
```

Note that any technology that supports SOAP can generate its own client classes with an automatic process, starting from the published web service descriptor (WSDL).

In the end, with the SOAP approach, we can keep a simple and neat code at both ends of the communication process (the web service implementation and the end client), while the hard work is done by the intermediate auto-generated classes.

**RPC and Document Based-WS: How to Communicate, Pros and Cons of the Two Approach** We have just seen how SOAP can leverage the developer's work by doing the entire hard job behind the scenes. Indeed, we have not even seen the content of the XML documents that client and server are exchanging. Well, this can be done using a TCP/IP monitor utility (for example, Apache TCMon).

Monitoring the request content of the client call gives the following results:

**Example 3.88—SOAP XML Request**

```
<?xml version="1.0" ?>
<soapenv:Envelope
    xmlns:soapenv="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/2001/XMLSchema"
    xmlns:ns1="http://item.service.soajava.packt.com/">
    <soapenv:Body>
        <ns1:insert>
            <arg0>
                <code>XY</code>
                <description>xy desc</description>
                <id>26</id>
            </arg0>
            </ns1:insert>
        </soapenv:Body>
    </soapenv:Envelope>
```

while the response content is as follows:

**Example 3.89—SOAP XML Response**

```
<?xml version="1.0" ?>
<S:Envelope xmlns:S="http://schemas.xmlsoap.org/soap/envelope/">
    <S:Body>
        <ns2:insertResponse
            xmlns:ns2="http://item.service.soajava.packt.com/">
            <return>
                <retCode>OK</retCode>
            </return>
        </ns2:insertResponse>
    </S:Body>
</S:Envelope>
```

```

<retMessage>Item was inserted successfully</retMessage>
</return>
</ns2:insertResponse>
</S:Body>
</S:Envelope>

```

As you can see, the XML content of a SOAP message is structured as an envelope which contains a mandatory body, while the header is optional. The content of the body element represents the payload that is the exchanged XML document.

However, when dealing with SOAP, a wide range of options are available. In our last example, we just adopted the default settings to keep things simpler and straightforward.

You may have heard, for example, about **binding style** (RPC or Document) or **use** (Encoded or Literal) or **parameter style** (Bare or Wrapped). In this paragraph, we will explore these concepts with particular emphasis on the binding style.

Before getting into this analysis though, we should spend some time on **WS-I**. The term stands for **Web Service—Interoperability** and represents a set of standards put together in order to allow the process of exchanging data throughout web services in a heterogeneous environment (for example, between Java and .NET). Of all the combinations of binding style, use, and parameter style, only the following are WS-I compliant, and we will concentrate exactly on them:

- RPC / literal
- Document / literal (bare or unwrapped)
- Document / literal wrapped

The "encoded" value for the **use** attribute is prohibited by WS-I. With this value, in fact, the data is serialized following the SOAP encoding described in Section 5 of SOAP 1.1 specification. Validating a SOAP encoded message against a WSDL description is quite a hard work, and since the validation is a fundamental step toward interoperability, only the use "literal" is allowed by WS-I.

**RPC/Literal.** One of the first architectural choices that has to be made when we decide to develop SOAP web services is whether to use RPC or Document binding style. **Remote Procedure Call (RPC)** is a generic mechanism throughout which is a procedure that resides on a computer (or a virtual machine) can be called by a program running on a different computer (or virtual machine). This paradigm has been around for decades and was implemented by several technologies, among which, the most popular are CORBA, DCOM, and RMI.

Despite the changes in technologies, an RPC call is always characterized by:

- A remote address
- A method (or operation) name
- A sequence of parameters
- A synchronous response

Note that, aside from the first, it shares the same characteristics of a classic local method call.

What does this old RPC paradigm have to do with SOAP and web services? Well, quite a lot indeed. In fact, in the early days of its definition (before being publicly published), SOAP was designed to support only RPC. It was, in a sense, a standardized evolution of the various distributed programming technologies.

With some modification in the annotation of the web service we designed last, we can switch it to RPC style (the JAX-WS default is "Document") and begin to explore this approach.

### **Example 3.90—SOAP RPC Style**

```

@WebService
@SOAPBinding(style=SOAPBinding.Style.RPC)
public class ItemWs {
    @WebMethod
    public Outcome insert(@WebParam(name="itemParam") Item item,
                          @WebParam(name="categoryParam") String category)
    {

```

```
//Insert item ...
```

As you can see we have introduced another parameter, the category, and our goal now is to insert an item into the specified category. We used the @WebParam annotation to give a name to each method argument.

Now, let's publish the service (just run the class with the Endpoint.publish line, the wsgen utility is not required in this case), import the client classes from the published WSDL with wsimport utility, and make a client call:

```
Outcome outcome = itemWs.insert(item1, "A");
```

If we monitor the request XML document, we will see the following structure:

**Example 3.91—SOAP RPC Request**

```
<soapenv:Body>
  <ans:insert xmlns:ans="http:// ... ">
    <itemParam>
      <code>XY</code>
      <description>xy desc</description>
      <id>26</id>
    </itemParam>
    <categoryParam>A</categoryParam>
  </ans:insert>
</soapenv:Body>
```

where we can recognize the typical RCP parts, that are the method name and the sequence of parameters.

The correspondent WSDL (that may be seen throughout the TCP monitor or pointing the browser to the URL ([http://localhost:8001/SoaBookSOAP\\_RPC\\_server/itemWs?WSDL](http://localhost:8001/SoaBookSOAP_RPC_server/itemWs?WSDL)) is listed here:

**Example 3.92—SOAP RPC WSDL**

```
<types>
  <xsd:schema>
    <xsd:import schemaLocation="http://127.0.0.1:8002/
      SoaBookSOAP_RPC_server/itemWs?xsd=1"
      namespace="http://item.service.
      soap.soajava.packt.com/"></xsd:import>
  </xsd:schema>
</types>
<message name="insert">
  <part name="itemParam" type="tns:item"></part>
  <part name="categoryParam" type="xsd:string"></part>
</message>
<message name="insertResponse">
  <part name="return" type="tns:outcome"></part>
</message>
<portType name="ItemWs">
  <operation name="insert" parameterOrder=
    "itemParam categoryParam">
    <input message="tns:insert"></input>
    <output message="tns:insertResponse"></output>
  </operation>
</portType>
<binding name="ItemWsPortBinding" type="tns:ItemWs">
  <soap:binding style="rpc"
    transport="http://schemas.xmlsoap.org/soap/http">
  </soap:binding>
  <operation name="insert">
    <soap:operation soapAction=""></soap:operation>
    <input>
      <soap:body use="literal" namespace=
        "http://item.service.soajava.packt.com/">
    </soap:body>
  </input>
  <output>
```

```

        <soap:body use="literal" namespace=
            "http://item.service.soajava.packt.com/">
        </soap:body>
    </output>
</operation>
</binding>
<service name="ItemWsService">
    <port name="ItemWsPort" binding="tns:ItemWsPortBinding">
        <soap:address location=
            "http://127.0.0.1:8002/SoaBookSOAP_RPC_server/itemWs">
        </soap:address>
    </port>
</service>
</definitions>

```

The schema location (<xsd:schema> block) is imported from URL:

[http://127.0.0.1:8001/SoaBookSOAP\\_RPC\\_server/itemWs?xsd=1](http://127.0.0.1:8001/SoaBookSOAP_RPC_server/itemWs?xsd=1)

and its content is the following:

#### **Example 3.93—SOAP RPC XSD**

```

<?xml version="1.0" encoding="UTF-8"?>
<xss:schema xmlns:tns="http://item.service.soajava.packt.com/"
  xmlns:xs="http://www.w3.org/2001/XMLSchema"
  targetNamespace="http://item.service.soajava.packt.com/" version="1.0">
    <xs:element name="Item" type="tns:item"></xs:element>
    <xs:element name="Outcome" type="tns:outcome"></xs:element>
    <xs:complexType name="item">
        <xs:sequence>
            <xs:element name="code" type="xs:string"
                minOccurs="0"/></xs:element>
            <xs:element name="description" type="xs:string"
                minOccurs="0"/></xs:element>
            <xs:element name="id" type="xs:int"></xs:element>
        </xs:sequence>
    </xs:complexType>
    <xs:complexType name="outcome">
        <xs:sequence>
            <xs:element name="retCode" type="xs:string"
                minOccurs="0"/></xs:element>
            <xs:element name="retMessage" type="xs:string"
                minOccurs="0"/></xs:element>
        </xs:sequence>
    </xs:complexType>
</xss:schema>

```

What should be noted here is that the schema defines only the complex type parameters (Item and Outcome in our case). It does not give any information useful to validate either the other simple parameters, or the rest of the SOAP message. Therefore, a major problem in adopting RPC style is that, the exchanged XML documents cannot be validated against an **XML Schema Definition (XSD)**.

Let's explore the Document style and see if it overcomes this limit.

**Document/Literal.** This style is also known as Document "bare" or "unwrapped" and we will soon get into the explanation of this term. For the moment the thing to pay attention to is that, using JAX-WS, the default value for the parameter style is "wrapped". Hence, in order to use the Document bare style, we have to set it explicitly.

#### **Example 3.94—SOAP Document style**

```

@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
            parameterStyle=SOAPBinding.ParameterStyle.BARE)
public class ItemWs {
    @WebMethod
    public Outcome insert(@WebParam(name="itemParam") Item item,
                          @WebParam(name="categoryParam") String category) {
        ...

```

Now, if we follow the usual steps in order to call this service from a client (again skipping the wsigen step), we will get an error while importing the client classes from the published WSDL.

### **error: operation "insert": more than one part bound to body**

This is an error that will indeed help us understand the difference between an RPC and a Document approach. What we should know is that WS-I only allows one child in the body of a SOAP message. But what is the reason of such a specification?

The Document style represents a new and different paradigm: the service input is "a document", not a request of the execution of a method with the correspondent parameter value. This means that a single object should be passed, and this object will be the sole input to the web service. The document will contain the information needed to perform its processing, but there is nothing here like a method name or a sequence of parameters. That is the reason why WS-I only allows one child.

Therefore, in order to perform our task (inserting an item into a category) with a Document, using WS-I compliant approach, we should refactor the service. We should create a new object called, for instance, ItemInsertRequest, which *wraps* the needed information (the item and the category). That is the reason of the name of this style (bare or unwrapped): there is no wrapping object around the parts; it must be created explicitly.

#### **Example 3.95—Request Wrapper**

```
@XmlRootElement(name = "ItemInsertRequest")
public class ItemInsertRequest {
    private Item item;
    private String category;
    ...
}
```

and the web service refactorized in order to have just one parameter (the document representing the request):

#### **Example 3.96—Web Service Using the Defined Wrapper**

```
@WebMethod
public Outcome insert(@WebParam(name="itemInsertRequestParam")
    ItemInsertRequest itemInsertRequest) {
    ...
}
```

In these conditions we will not get errors while generating the client classes with the wsimport utility, and can finally make the refactorized client call:

#### **Example 3.97—SOAP Document Client Request**

```
ItemInsertRequest req = new ItemInsertRequest();
req.setItem(item1);
req.setCategory("A");
Outcome outcome = itemWs.insert(req);
```

that will be forwarded with the following SOAP body:

#### **Example 3.98—SOAP Document XML Request**

```
<ns1:itemInsertRequestParam>
    <category>A</category>
    <item>
        <code>XY</code>
        <description>xy desc</description>
        <id>26</id>
    </item>
</ns1:itemInsertRequestParam>
```

Please note that, although the structure is indeed the same as with RPC style (see listing 19), we are now dealing with a *document* instead of a method call while the category and item are no more parameters, but just *attributes* of this document.

What should be noted instead about the WSDL is that, other than having *just one part* inside the input message and having the style set to *document*, the attribute *type* has gone missing and a correspondent *element* attribute has taken its place.

### **Example 3.99—SOAP Document WSDL**

```
<message name="insert">
    <part element="tns:itemInsertRequestParam"
          name="itemInsertRequestParam"></part>
</message>
...
<binding name="ItemWsPortBinding" type="tns:ItemWs">
    <soap:binding style="document"
```

As far as the XML schema is concerned, it can now be used to validate the entire document:

### **Example 3.100—SOAP Document XSD**

```
<xs:element name="Item" type="tns:item"></xs:element>
<xs:element name="ItemInsertRequest" type="tns:itemInsertRequest"></xs:element>
<xs:element name="Outcome" type="tns:outcome"></xs:element>
<xs:element nillable="true" name="insertResponse"
            type="tns:outcome"></xs:element>
<xs:element nillable="true" name="itemInsertRequestParam"
            type="tns:itemInsertRequest"></xs:element>

<xs:complexType name="itemInsertRequest">
    <xs:sequence>
        <xs:element name="category" type="xs:string" minOccurs="0"></xs:element>
        <xs:element name="item" type="tns:item" minOccurs="0"></xs:element>
    </xs:sequence>
</xs:complexType>

<xs:complexType name="item">
    <xs:sequence>
        <xs:element name="code" type="xs:string" minOccurs="0"></xs:element>
        <xs:element name="description" type="xs:string" minOccurs="0"></xs:element>
        <xs:element name="id" type="xs:int"></xs:element>
    </xs:sequence>
</xs:complexType>
```

The main strength of Document/literal style is therefore the ability to allow the validation of the whole XML document exchange.

With this approach, though, we have lost something the operation name is no more present in the SOAP message. This may be a drawback in some situations, take for instance the case where the message is transmitted over an asynchronous TCP/IP protocol such as SMTP. The process of dispatching the message may be difficult, if not impossible.

Another disadvantage of this style is that, if we are dealing with already developed applications, a certain effort has to be taken into account in order to refactor both the server and the client side code.

**Document/Literal Wrapped.** This style is the default in JAX-WS, and we have already made use of it in our very first SOAP example. The following annotation is in fact useless with JAX-WS:

### **Example 3.101—SOAP Document Wrapped Style**

```
@SOAPBinding(style=SOAPBinding.Style.DOCUMENT,
              parameterStyle=SOAPBinding.ParameterStyle.WRAPPED)
```

Now, let's go back to the initial structure of the web service (before the refactoring needed by the Document bare style):

### **Example 3.102—SOAP Document Wrapped Web Service**

```
@WebMethod
public Outcome insert(@WebParam(name="itemParam") Item item,
                      @WebParam(name="categoryParam") String category) {
```

and follow the usual steps in order to make a client call (now the wsigen step is required).

Well, now we will not get any error, even if we have more than one parameter, as in RPC style. What does this style do to adhere to the Document style, without forcing us to refactor our source code? It simply *wraps*

(automatically, without our effort) the method name and the parameters into a new object, whose name is the same of the method itself.

In fact, the body of the SOAP message is now in this form:

**Example 3.103—SOAP Document Wrapped XML Request**

```
<ns1:insert>
  <itemParam>
    <code>XY</code>
    <description>xy desc</description>
    <id>26</id>
  </itemParam>
  <categoryParam>A</categoryParam>
</ns1:insert>
```

In conclusion, the Document / literal wrapped style gathers the advantages from both Document and RPC approaches:

- The SOAP message can be validated against an XML schema
- The SOAP body contains only one child, and is thus WS-I compliant
- Multiple parameters are allowed without any refactoring
- The operation name is contained in the message

It is, in fact, the default when we use JAX-WS and in general a good choice for most cases.

**A Hello Example.** In this section, you will not use any application server or any third-party web server, but will use just Java SE 6 and its tools to develop and deploy a simple web service.

The server is composed of three Java explained below:

**IHello** is a java interface and is shown here:

```
public interface IHello{
    String sayHello (String name);
}
```

**HelloImpl** implements the business functionality to be exposed as web service. This class realizes the preceding IHello interface:

```
import javax.jws.WebService;
import javax.jws.soap.SOAPBinding;
import javax.jws.WebMethod;

@WebService(name="IHello", serviceName="HelloService")
@SOAPBinding(style=SOAPBinding.Style.RPC)
public class HelloImpl implements IHello{
    @WebMethod(operationName = "sayHello")
    public String sayHello(String name){
        System.out.println("HelloImpl.sayHello...\"");
        return "\nHello From Server !! : " + name;
    }
}
```

HelloImpl is annotated with javax.jws.WebService annotation. The @WebService annotation defines the class as a web service endpoint. The javax.jws.soap.SOAPBinding annotation specifies the mapping of the web service onto the SOAP message protocol. HelloImpl declares a single method named sayHello, which is annotated with the @WebMethod annotation. This annotation will expose the annotated method to web service clients. In fact, the IHello interface is not required while building a JAX-WS endpoint, but we have used it here as a good programming practice.

**HelloServer** is a main class which makes use of javax.xml.ws.Endpoint for publishing the web service:

```
import javax.xml.ws.Endpoint;
```

```

public class HelloServer {
    public static void main(String args[]) {
        log("HelloServer.main : Creating HelloImpl...");
        IHello iHello = new HelloImpl();
        try{
            // Create and publish the endpoint at the given address
            log("HelloServer.main : Publishing HelloImpl...");
            Endpoint endpoint1 =
                Endpoint.publish("http://localhost:8080/Hello", iHello);
            log("HelloServer.main : Published Implementor...");
        }
        catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
        System.out.println("HelloServer Exiting ...");
    }
}

```

The Client is composed of one Java file. The **HelloClient** is dependent on two auto generated classes for example, HelloService and IHello. These classes will be auto generated, when we will build the client later in this exercise. The following code of client is straightforward:

```

public class HelloClient{
    public static void main(String args[]) {
        log("HelloClient.main : Creating HelloImpl...");
        HelloService helloService = null;
        IHello helloImpl = null;
        String gotFromServer = null;

        try{
            log("HelloClient.main : Creating HelloImplService...");
            if(args.length != 0){
                helloService = new HelloService(new URL(args[0]),
                    new QName(args[1], args[2]));
            }
            else{
                helloService = new HelloService();
            }
            log("HelloClient.main : Retreiving HelloImpl...");
            helloImpl = helloService.getIHelloPort();
            log("HelloClient.main : Invoking
                helloImpl.sayHello(\"Binil\")...");
            gotFromServer = helloImpl.sayHello("Binil");
            log("HelloClient.main : gotFromServer : " + gotFromServer);
        }
        catch (Exception e) {
            System.err.println("ERROR: " + e);
            e.printStackTrace(System.out);
        }
    }
}

```

You will first have to instantiate the HelloService, which has the required plumbing to connect to the web service. Then, you will get a reference to the port using which you can invoke the remote web service.

When we build the client, we also auto generate some client side artifacts out of the deployed web service using the following ant task:

```

<target name="GenSrc">
    <exec executable="${env.JAVA_HOME}/bin/wsimport">
        <arg line="-keep
            -d build
            -p com.binildas.ws.javastandalone.simple

```

```

        -s ${gensrc}  http://localhost:8080/Hello?WSDL"/>
    </exec>
</target>
```

The client code is dependent on these generated files. So, we can now build the client codebase, and then send a web service request to the server. Any response received from the server is printed to the console.

Let us now move on to an **Enterprise Server** and deploy a similar web service there.

**HelloWebService** is again an annotated java class. The annotations have the same meaning as in the earlier sample.

```

@WebService
public class HelloWebService{
    private static int times;

    public HelloWebService(){
        System.out.println("Inside HelloWebService.HelloWebService...");
```

```

    public String hello(String param){
        System.out.println("Inside HelloWebService.hello... - " + (++times));
        return "Return From Server : Hello " + param;
    }
}
```

The code for the **client** is very simple and is shown here:

```

public class Client{

    @WebServiceRef(wsdlLocation = "http://localhost:8080/
        HelloWebService/HelloWebServiceService?WSDL")
    static HelloWebServiceService service;

    public static void main(String[] args){
        Client client = new Client();
        client.test();
    }

    public void test(){
        try{
            HelloWebService helloWebServicePort = service.getHelloWebServicePort();
            String ret =
                helloWebServicePort.hello(System.getProperty("user.name"));
            System.out.println("Hello result = " + ret);
        }
        catch(Exception e){
            e.printStackTrace();
        }
    }
}
```

Here, we use the javax.xml.ws.WebServiceRef annotation to declare a reference to the deployed web service. @WebServiceRef uses the wsdlLocation element to specify the URI of the HelloWebService's WSDL file. Then, the client gets a proxy to the remote web service and invokes the web service method.

The WSDL for the deployed web service would be available in the URL

```
http://localhost:8080/HelloWebService/HelloWebServiceService?WSDL
```

### 3.3.11 JXTA

Although all peer-to-peer systems are distributed systems, not all distributed systems aim to facilitate peer-to-peer-style computing. Growing experience with real-life P2P computing indicates that peer-to-peer systems share a common set of needs above and beyond what a general purpose distributed computing framework needs to provide. JXTA is an attempt to factor out the needs common to peer-to-peer systems, and offer a set of APIs around those functionalities.

In essence, JXTA is a peer-to-peer developer's library. It enables a developer to delegate to it the grunt work involved in the P2P-specific aspects of an application, and to focus instead on what makes his application unique.

#### 3.3.11.1 The JXTA Protocols

JXTA's designers chose to define the operations of the JXTA virtual network in terms of a set of protocols. A protocol, in essence, is a form of agreement in support of an activity. As long as all parties to an activity understand and follow the required protocols, the activity can take place. A corollary to that statement is that those protocols are the only thing each party has to understand to participate in the desired action.

In programming terminology, this means that if two peers want to participate on the JXTA network, the only thing they must have in common is an understanding of the protocols required for that participation. As far as JXTA is concerned, the peers might be implemented in different programming languages, run on different types of machines, or transmit messages through different network transport layer protocols (for example, TCP, HTTP, or Bluetooth). As long as the peers understand the JXTA protocols, they can become part of the JXTA virtual network and communicate with each other.

The JXTA protocols are expressed in terms of message exchanges. Regardless of what programming language a peer uses, at some point in the communication process, it must translate—or bind—JXTA messages to constructs in that programming language. Currently, several JXTA programming language bindings—or JXTA bindings, for short—exist or are being developed for Java, C, Objective C, Perl, Python, and Smalltalk, just to mention a few. The JXTA community actively develops new language bindings, so this list is growing.

From a programmer's viewpoint, a peer interacts with the JXTA system via an API provided by the JXTA bindings to the peer's programming language. An implementation of that API exposes the core functionality defined by the JXTA protocols as services a client can use to participate in P2P interactions. Because this book is about peer-to-peer programming in Java, we will only discuss the Java API to JXTA; other books and online resources provide descriptions of programming JXTA in other programming languages. Although the APIs to the JXTA protocols are different from programming language to programming language—and it is indeed possible to design different APIs for a single language—the JXTA protocols are defined via their message formats alone, and without reference to any specific programming language API.

The protocol specifications define JXTA messages in terms of XML data structures, as XML is a de facto standard for cross-platform representation of structured data. However, JXTA does not require that a peer have full XML-processing capability. Peers with limited resources, for example, might choose to precompile JXTA protocol exchanges into a binary representation. As long as those messages conform to the protocol specifications, that peer is able to participate on the JXTA network without having to process XML.

Although the JXTA specifications currently define seven protocols, a peer is not required to understand all seven to become part of the JXTA virtual network. Rather, the protocols define how a peer should act if it decides to implement the behavior defined by a protocol. The more protocols a peer supports, the fuller its participation in the JXTA network. In addition, a peer can also extend any of the existing protocols with new behavior.

**Peer Resolver Protocol.** The JXTA protocol messages form sets of query-response pairs: A peer sends a query, and some other peer sends a response to that query. The format of the query and response messages are what the protocol specification defines. Responses to a query occur asynchronously in JXTA. The Peer Resolver Protocol (PRP) is the most fundamental JXTA protocol that specifies how a query is paired up with one or more responses.

A query message is directed to a query handler. A peer that implements that handler might choose to process the query and respond. Processing a query can include any computation on the peer offering the query handler, and that computation might consume an arbitrary query string or document. Similarly, the processing of the query may result in an arbitrary string. Responses are matched up with queries by a service providing the PRP.

All other protocols needing a generic query-response mechanism rely on PRP. It helps standardize the formats of queries and query resolution. An implementation of PRP offers a generic query service to the JXTA network. A query message is addressed not to a specific peer, but to a named query handler. A peer registers a query handler, and it can then answer queries addressed to that handler. Note, however, that a peer is not obligated to answer a query; also, answers as well as queries might be lost on the network because of network or peer failure.

The following XML schema defines the resolver query message:

```
<xs:element name="ResolverQuery" type="jxta:ResolverQuery"/>
<xs:complexType name="ResolverQuery">
    <xs:element name="Credential" type="xs:anyType" minOccurs="0"/>
    <xs:element name="SrcPeerID" type="JXTAID"/>
    <xs:element name="HandlerName" type="xs:string"/>
    <xs:element name="QueryID" type="xs:string" minOccurs="0"/>
    <xs:element name="Query" type="xs:anyType"/>
</xs:complexType>
```

As the schema definition shows, a query includes the sending peer's credential and peer ID, the name of the handler for the query, a unique query ID, a chunk of free text that forms the body of the query. A peer that registers as a handler for the query can use the query text to perform some processing to determine a response.

The response message format is as follows:

```
<xs:element name="ResolverResponse" type="ResolverResponse"/>
<xs:complexType name="ResolverResponse">
    <xs:element name="Credential" type="xs:anyType" minOccurs="0"/>
    <xs:element name="HandlerName" type="xs:string"/>
    <xs:element name="QueryID" type="xs:string" minOccurs="0"/>
    <xs:element name="Response" type="xs:anyType"/>
</xs:complexType>
```

The response contains the credential of the responding peer, as well as a text element Response. We will shortly explore the API offered by the Java bindings to take advantage of the PRP.

**The Query-Response Paradigm.** JXTA requires two-way communication between peers—a query is of little importance if the sender is not able to receive a reply because of network limitations. Although this requirement appears easy to satisfy, some networks, such as pager networks, support only one-way communication (those are not able to participate in the PRP).

Because the PRP is a fundamental JXTA protocol in the sense that other protocols depend on it for query/response resolution, such limited peers are not able to participate in the JXTA virtual network. However, the requirement to support two-way communication does not exclude networks with highly asymmetric characteristics. For example, it might not be possible to directly contact a peer inside a firewall, but a network might provide a mechanism to contact that peer in some indirect way, such as through a gateway. As long as that is possible, the peer inside the firewall can participate in the JXTA protocols with peers outside of it.

**The Endpoint Routing Protocol (ERP)** specifies message routing in JXTA. We earlier described how routing consists of constructing an ordered list of hops through which a message travels from a peer to its desired destination. Relay peers that offer routing capability perform this task. If a router receives a route query and knows the route to the requested destination, it answers that route request with a list of hops that constitute the route. The sender of that message will use the first address in that list and dispatch the message to it. Network changes can cause a route to become obsolete at any time during the message's transmission, necessitating the discovery of a new route. The following XML schema defines the structure of a route:

```
<?xml version="1.0" encoding="UTF-8"?>
```

```

<jxta:EndpointRouter>
    <Src> peer id of the source </Src>
    <Dest> peer id of the destination </Dest>
    <TTL> time to live </TTL>
    <Gateway> ordered sequence of gateway </Gateway>
    <.....>
    <Gateway> ordered sequence of gateway </Gateway>
</jxta:EndpointRouter>

```

As the message winds its way through the network toward its destination, each peer it goes through leaves a trace on that message. Those traces enable routers to remember new routes to destinations. As more peers cache correct route information to a destination, the faster route discovery becomes. The trace information left by peers on a message is also useful to detect loops, or to detect duplicate messages by routers.

The following XML schema defines the route query message, and shows how route caching is controlled:

```

<?xml version="1.0" encoding="UTF-8"?>
<jxta:EndpointRouterQuery>
    <Credential> credential </Credential>
    <Dest> peer id of the destination </Dest>
    <Cached>
        true: if the reply can be a cached reply
        false: if the reply must not come from a cache
    </Cached>
</jxta:EndpointRouterQuery>

```

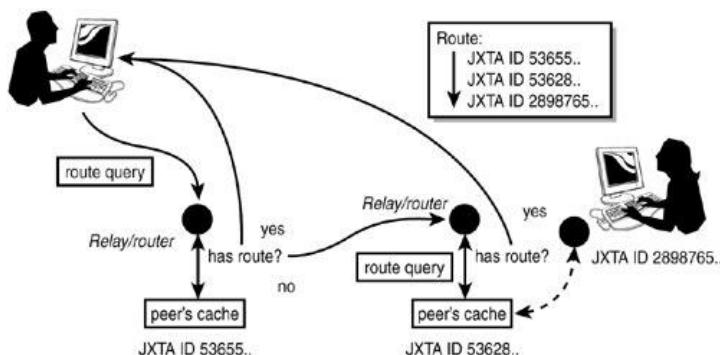
In response to this message, a peer router that has the requested route information returns an answer conforming to the following format:

```

<?xml version="1.0" encoding="UTF-8"?>
<jxta:EndpointRouterAnswer>
    <Credential> credential </Credential>
    <Dest> peer id of the destination </Dest>
    <RoutingPeer> Peer ID of the router that knows a route to DestPeer
    </RoutingPeer>
    <RoutingPeerAdv> Advertisement of the routing peer </RoutingPeerAdv>
    <Gateway> ordered sequence of gateway </Gateway>
    <.....>
    <Gateway> ordered sequence of gateway </Gateway>
</EndpointRouterAnswer>

```

By defining this somewhat cumbersome protocol, ERP's objective was to ensure a high degree of success in guiding a message to its destination, not to ensure efficiency in message transmission. Thus, more intelligent peers might implement specialized routing algorithms to optimize route discovery. Figure 3.50 illustrates ERP in action.



**Fig. 3.50. ERP discovers routes and aids in message delivery.**

**Peer Discovery Protocol.** You might recall from earlier in this chapter that advertisements provide the mechanism to describe resources on the JXTA virtual network. The discovery of resources based on their advertisements is what the Peer Discovery Protocol (PDP) specifies. The query message for PDP offers three pieces of information:

- Advertisement type
- An XML key or tag name
- A value that must correspond to the XML key

The message format is specified as follows:

```
<xs:element name="DiscoveryQuery" type="jxta:DiscoveryQuery"/>
<xs:complexType name="DiscoveryQuery">
    <xs:element name="Type" type="xs:string"/>
    <xs:element name="Threshold" type="xs:unsignedInt" minOccurs="0"/>
    <xs:element name="PeerAdv" type="xs:string" minOccurs="0"/>
    <xs:element name="Attr" type="xs:string" minOccurs="0"/>
    <xs:element name="Value" type="xs:string" minOccurs="0"/>
</xs:complexType>
```

The threshold attribute in the query message defines how many responses per peer we are expecting. The response message is as follows:

```
<xs:element name="DiscoveryResponse" type="jxta:DiscoveryResponse"/>
<xs:complexType name="DiscoveryResponse">
    <xs:element name="Type" type="xs:string"/>
    <xs:element name="Count" type="xs:unsignedInt" minOccurs="0"/>
    <xs:element name="PeerAdv" type="xs:anyType" minOccurs="0">
        <xs:attribute name="Expiration" type="xs:unsignedLong"/>
    </xs:element>
    <xs:element name="Attr" type="xs:string" minOccurs="0"/>
    <xs:element name="Value" type="xs:string" minOccurs="0"/>
    <xs:element name="Response" type="xs:anyType" maxOccurs="unbounded">
        <xs:attribute name="Expiration" type="xs:unsignedLong"/>
    </xs:element>
</xs:complexType>
```

The response element includes the advertisement that the JXTA peer located in response to the query, in addition to the responding peer's own advertisement. The message exchange defined by PDP represents the lowest-level discovery mechanism in JXTA. Applications can build higher-level discovery protocols based on PDP.

**Rendezvous Protocol.** By default, query messages only reach peers that share a physical network with the inquiring peer. The Rendezvous Protocol (RP) defines how queries propagate through rendezvous peers. The propagation scope for messages via the RP is limited to a set of peers that form a logical group, or peer group. (We will discuss peer groups shortly.) This scope is reflected in the advertisement message of a peer that announces itself as a rendezvous peer, as the following XML message from the JXTA specifications shows:

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:RdvAdvertisement>
    <Name> name of the rendezvous peer</Name>
    <RdvGroupId> PeerGroup UUID </RdvGroupId>
    <RdvPeerId>Peer ID of the rendezvous peer</RdvPeerId>
</jxta:RdvAdvertisement>
```

A peer discovers a rendezvous peer via this advertisement type and then connects to it. When a peer requests that connection, the rendezvous peer assigns a lease to the connection; the request for the connection includes the desired lease time. A rendezvous peer receives a message and propagates that message to other rendezvous peers it knows. The message format that controls that propagation assumes the following format:

```
<xs:element name="RendezVousPropagateMessage"
    type="jxta: RendezVousPropagateMessage"/>
<xs:complexType name="RendezVousPropagateMessage">
    <xs:element name="MessageId" type="xs:string"/>
    <xs:element name="DestSName" type="xs:string"/>
    <xs:element name="DestSPParam" type="xs:string"/>
    <xs:element name="TTL" type="xs:unsignedInt"/>
    <xs:element name="Path" type="xs:anyURI" maxOccurs="unbounded"/>
```

```
</xs:complexType>
```

**Pipe Binding Protocol.** A pipe defines a communication channel between two peers. Each pipe has two ends: an input pipe and an output pipe, for receiving and sending messages through the pipe, respectively. The Pipe Binding Protocol (PBP) defines how a pipe's input and output ends bind to a peer's endpoint. JXTA defines different types of pipes, and a pipe's advertisement includes an indication of the pipe's type, in addition to its name and ID:

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PipeAdvertisement>
    <Name> name of the pipe</Name>
    <Id> Pipe Id </Id>
    <Type> Pipe Type </Type>
</jxta:PipeAdvertisement>
```

The following message format defines a pipe binding request message:

```
<xs:element name="PipeResolver" type="jxta:PipeResolver"/>
<xs:complexType name="PipeResolver">
    <xs:element name="MsgType" type="xs:string"/>
    <xs:element name="PipeId" type="JXTAID"/>
    <xs:element name="Type" type="xs:string" minOccurs="0"/>
    <xs:element name="Cached" type="xs:boolean" default="false" minOccurs="0"/>
    <xs:element name="Peer" type="JXTAID" minOccurs="0"/>
    <xs:element name="Found" type="xs:boolean" minOccurs="0"/>
    <xs:element name="PeerAdv" type="xs:string" minOccurs="0"/>
</xs:complexType>
```

The Found and PeerAdv elements in this message are sent by the peer that was able to resolve the pipe's endpoint. We will see shortly how pipe endpoint resolution occurs via the Java API.

**The Peer Information Protocol (PIP)** fills a bit of a different role in the JXTA universe than the other protocols. Its purpose is to allow peers to exchange runtime information, such as a peer's uptime, or the number of messages a peer processed in a given period of time. That PIP exists in the JXTA specifications reveals the importance of such metadata about peers for the proper functioning of a P2P network. Tool vendors can use this protocol to hook network-monitoring software into peers and allow system administrators to track the performance of the JXTA network. As it exists in the JXTA specifications, PIP is rather limited in the kinds of information it defines about a peer. In addition, some peers that claim to support PIP might only provide a partial set of that information. The idea behind PIP, though, is that implementers of this protocol can extend it with other sorts of information; because this protocol, as every other JXTA protocol, uses XML to define its message formats, extending PIP with new information is as easy as adding elements to an XML document. As it stands now, PIP's message format is:

```
<xs:element name="PeerInfoResponse" type="jxta:PeerInfoResponse"/>
<xs:complexType name="PeerInfoResponse">
    <xs:element name="sourcePid" type="xs:anyURI"/>
    <xs:element name="targetPid" type="xs:anyURI"/>
    <xs:element name="uptime" type="xs:unsignedLong" minOccurs="0"/>
    <xs:element name="timestamp" type="xs:unsignedLong" minOccurs="0"/>
    <xs:element name="response" type="xs:anyType" minOccurs="0"/>
    <xs:element name="traffic" type="jxta:piptraffic" minOccurs="0"/>
</xs:complexType>
<xs:complexType name="piptraffic">
    <xs:element name="lastIncomingMessageAt" type="xs:unsignedLong"
minOccurs="0"/>
    <xs:element name="lastOutgoingMessageAt" type="xs:unsignedLong"
minOccurs="0"/>
    <xs:element name="in" type="jxta:piptrafficinfo" minOccurs="0"/>
    <xs:element name="out" type="jxta:piptrafficinfo" minOccurs="0"/>
</xs:complexType>
<xs:complexType name="piptrafficinfo">
    <xs:element name="transport" type="xs:unsignedLong" maxOccurs="unbounded">
        <xs:attribute name="endptaddr" type="xs:anyURI"/>
    </xs:element>
</xs:complexType>
```

**Peer Groups and the Peer Membership Protocol.** One of the most powerful aspects of JXTA is its capability to overcome the physical partitioning of a network. Peers can communicate on the JXTA network, even though they might reside on different sides of firewalls and NAT points, or use different network communication protocols. However, network partitioning in the physical world is useful for various reasons. For example, by using a set of private IP addresses behind a secure NAT point, a company's computers form a group inaccessible from the outside. The company can provide services on that private network, such as databases or application software, without having to be concerned about unauthorized access from the outside. JXTA offers the equivalent of network partitions in the form of peer groups.

Any sort of peer group can be created, each with a unique ID and a name. As other JXTA entities with IDs, each peer group has an advertisement. Peers that share a common interest join a peer group, and a peer may be a member of one or more groups. There are several reasons why a peer might want to become member of a group.

First, groups form a scoping environment for JXTA queries: when a peer propagates a query for an advertisement, that query will by default propagate only through peers that are members of the peer group in which message propagation commenced. Because of query propagation scoping, resources available to one peer in a group are accessible to other group members as well, facilitating resource sharing. Such resources include anything with an advertisement, such as pipes or other peers. Because pipes and message propagation are the mechanisms through which peers communicate, peers must belong to the same peer group to send messages to one another.

Second, peer groups outline security boundaries on the JXTA network. When a peer wants to join a peer group, it must apply for membership in that group. The Peer Membership Protocol (PMP) defines how a peer obtains group membership.

In essence, the application is similar to someone applying for membership in a professional organization. An applicant initiates the process by filling out an application form with some basic information, such as job history and educational background. Because many professional organizations require a minimal education level or job history from candidates, the organization's membership committee then evaluates the application, possibly confirming the information the applicant provided. After that information has been verified, the applicant is qualified to join. At that point, the applicant joins by paying the initial membership dues and signing a membership form. Thereafter, the new member receives a membership card, verifying his standing as the organization's member.

The JXTA PMP outlines a similar protocol, consisting of an application phase and a joining phase. The process of joining a JXTA peer group proceed as follows. Each peer group operates a service implementing PMP. When a peer applies for group membership, it specifies its credentials, in addition to a reference to the authenticator capable of verifying those credentials. The group's membership service then uses that information to verify the peer's identity via the specified authenticator. That verification might take any form, including contacting third-party components, such as a centralized database, or asking a peer to answer an arbitrary set of questions. After the peer prequalifies for group membership in this manner, it can request to join the group. Joining results in a credential issued by the membership service. That credential will thereafter be used in all peer operations requiring group membership, such as accessing services and other resources specific to a group.

Finally, JXTA groups provide mechanisms for peers to monitor other peers sharing a peer group. In that way, JXTA peer groups define a monitoring environment as well.

The JXTA specifications define two kinds of peer groups: a Platform Peer Group and a Standard Peer Group. By default, every peer is a member of the Platform, or World Peer Group, and joins that group when the peer boots up. The World Peer Group defines implementations of the basic JXTA protocols, such as the membership, discovery, and resolver protocols. Peer groups form a parent-child hierarchy. All the services available in a parent group are also available in child groups.

By virtue of membership in the World Peer Group, every peer on the global JXTA network can potentially communicate with every other peer. The Standard Peer Group, in turn, can be used to implement user-defined peer groups (to provide a scoping, security, or monitoring environment).

### 3.3.11.2 JXTA Services

The JXTA protocols are definitions of abstract behavior, by means of the message formats that pass between peers. In order for the peers to participate in the JXTA virtual network, they must have implementations of those protocols available, and exposed via an application programmer's interface (API). JXTA protocol implementations are services. Services also include software components that support activities other than the core JXTA protocols,

and which are available for other peers to use. Those services rely on the core JXTA service implementations. Higher-level JXTA-based applications depend on an increasingly higher-level service layer.

As any other JXTA resource, services assume unique identifiers, and announce their presence via JXTA advertisements. If a service is capable of communicating with other peers via JXTA pipes, its advertisement might include the communication pipes' advertisements as well.

Because services have IDs and corresponding advertisements, service discovery is similar to peer discovery: A peer issues a query for a specific service. That query, in turn, propagates through the peer group via the rendezvous network until rendezvous peers locate an advertisement for the specified service. At that point, the service's advertisement is returned to the requestor, causing that advertisement to be cached by rendezvous peers along its way. The service advertisement could contain the advertisement of a pipe described in the service's advertisement. The requestor peer can invoke the service through that pipe.

Although pipes are the JXTA-specified means of peer-to-peer message passing, JXTA does not prescribe how a service should be invoked: Any service invocation is possible, including opening direct socket connections to the service, performing remote method invocations (RMI) on a remote service object, or simply sending messages to the target peer formatted in accord with the Simple Object Access Protocol (SOAP) document model.

To support flexible service invocation, some services might specify a proxy object: An object that represents the service on the JXTA network, and has the capability to communicate to other network resources, some of which might not be accessible via JXTA communication mechanism. This proxy-based service invocation is similar to Jini's service proxies, which are free to choose any communication mechanism to connect to other network resources.

JXTA provides for high service availability by having certain services belong, not to a single peer, but to the whole peer group. In essence, several peers implement the service, and those implementations are deemed equivalent by the JXTA system; that is, they have the same IDs and advertisements. A peer requesting a peer group service can use any of those service instances. The services implementing the core JXTA protocols are peer group services.

Note that JXTA does not provide for load balancing or service fail-over. If a peer providing a peer group service fails, the client of a service on the failed peer must discover a new instance of the service. If the failed peer contains data that was modified in the course of using that service, that data will not be available on other peers running instances of the service, unless an application developer built data replication into his JXTA service implementation.

### 3.3.11.3 JXTA Modules

In locating and interacting with other services, JXTA aims to facilitate peer-to-peer computing across a diversity of language and execution environments. A module denotes a chunk of functionality (code) available to JXTA peers in a peer group. JXTA uses the concept of module IDs and advertisements to define services and service types.

In JXTA, you can locate a service based on its abstract behavior, a particular specification or wire protocol for that service, or an implementation of a specification. An underlying assumption is that you have some prior knowledge to construct a query for any of those advertisements. That prior knowledge typically means knowing the IDs for a module's class, spec, or implementation, respectively.

An identifier for an API, or some local behavior, is the module's class ID, which also includes a base class ID, declaring what class this module specializes (extends). Services available in a group are defined by a set of module class IDs. Different groups might use specifications of common classes which are not network-compatible, but the local APIs for those classes will be similar, if their module class IDs are identical. Combining the peer group's ID with the module's class ID uniquely identifies a service.

When constructing a JXTA service, you would normally describe what other JXTA services your service depends on based on those services' module class IDs. To draw a comparison to Java classes, a module class is somewhat analogous to a Java interface—a Java interface also defines some local behavior, but how that behavior might be accessed over the network is specific to an implementation of that interface. (Keep in mind that JXTA module classes are not specified in the Java programming language, but by abstract IDs and advertisements.)

Module class IDs are advertised by module class advertisements. A module class advertisement provides a description of what a module class ID means. These advertisements are meant for programmers who want to

create modules offering the abstract behavior designated by the module's class ID. It does not specify how to invoke the service.

There can be many embodiments of a module class, each of which is termed a module specification, or module spec. A module specification represents network behavior, or wire protocols, that a module might embed. Module specs with identical Module Spec IDs are network-compatible.

A module spec advertisement provides the description of the protocol defined by the module spec. Although a module class defines a service's dependency on some abstract behavior, a module spec designates dependency on a given specification for that behavior. As such, module spec advertisements might also describe how to access a module. Similar to module class advertisement, module spec advertisements are also chiefly aimed at a programmer implementing a specification.

Using a module's spec ID, an implementation of that module can be located in the peer group. These implementations themselves are defined by module implementation IDs. Module implementation IDs are advertised via module implementation advertisements. This type of advertisement might provide a complete URI to the code and JAR files needed to execute the implementation, in addition to other descriptive elements; a peer could use those URIs to download the needed code. The module spec ID on which an implementation is based is also included.

Note that JXTA does not mandate that a module's implementation be available locally to a peer. Rather, using information in the module implementation advertisement, a peer can download pieces of code (JAR files) needed to instantiate and invoke a service. How a peer might download that code is not specified by JXTA—the peer could use HTTP, FTP, or any other network protocol. A peer might even use the Java mobile object paradigm specified in RMI.

### 3.3.11.4 The JXTA J2SE API

The main purpose of the J2SE API to JXTA is to allow Java programs to take advantage of the unique capabilities offered by this P2P platform. A secondary purpose of the J2SE implementation is to offer a test bed for the JXTA protocols, and to enable programmers to see how well those protocols serve real-life applications. Because APIs are for programmers, the J2SE API aims to simplify interaction with the JXTA protocols and core services from a Java programmer's viewpoint.

You can download the current version of the J2SE implementation from <http://www.jxta.org>. The only requirements for the JXTA J2SE binding is the presence of a J2SE implementation. Other Java-based JXTA implementations might require various configurations of J2ME.

### 3.3.11.5 First JXTA Program

Although the JXTA download comes with several sample applications that enable you to experiment with the platform, we will write here a very simple JXTA program that initializes the platform and displays some information about peers and peer groups. Successful termination of this program indicates that you have correctly installed the JXTA libraries. The following code shows the complete source code for this simple program that starts up the JXTA runtime environment and then prints out the peer and peer group names and IDs, respectively.

```
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.exception.PeerGroupException;

/*Platform test. This program simply initializes the JXTA platform,
 * obtains a reference to the NetPeerGroup, and displays the names and IDs
 * of the peer and the group, respectively. */
public class PlatformTest {
    public static void main(String[] argv) {
        PeerGroup netPeerGroup = null;
        try {
            netPeerGroup = PeerGroupFactory.newNetPeerGroup();
            System.out.println("Group name: " +
                netPeerGroup.getPeerGroupName());
            System.out.println("Group ID: " +
```

```

        netPeerGroup.getPeerGroupID());
        System.out.println("Peer name: " +
                           netPeerGroup.getPeerName());
        System.out.println("Peer ID: " +
                           netPeerGroup.getPeerID());
    } catch (PeerGroupException e) {
        System.out.println("Can't initialize net peer group: " +
                           e.getMessage());
        System.exit(0);
    }
}

```

When you compile and run this program for the first time, the JXTA runtime brings up a window requesting that you configure your peer. This window represents the JXTA Configurator. It asks you to give your peer a name, and to indicate whether your peer will offer any relay services, such becoming a rendezvous, a router, or a gateway. You also need to specify whether to use any outside relays to aid the discovery of advertisements outside of your local network. You have the option of asking the system to download a list of well-known rendezvous and relays. Once you opt to download that list, your peer will use those in message propagation and discovery. Finally, you also need to specify a secure username and password for your peer. This information will be used when your peer applies for group membership, and in other situations when it needs to be authenticated.

As we mentioned earlier, at startup a peer initializes the World Peer Group and the Net Peer Group. This application simply obtains a reference to the latter, and prints out that peer group's name, group ID, as well as the peer's own name and ID:

```

aquinas% java PlatformTest
Group name: NetPeerGroup
Group ID: urn:jxta:jxta-NetGroup
Peer name: aquinas
Peer ID: urn:jxta:
           uuid59616261646162614A78746150325033D5A4650661A84DF1867897A0FC52E4FF03

```

If you examine the files in the directory you started the application from, you will see that the JXTA runtime created several files and subdirectories. The peer's configuration itself is stored in the file `PlatformConfig`; this is an XML file, and you can examine its contents with a text editor. If you ever need to reconfigure the peer, removing this file will cause the JXTA Configurator to display the set of configuration screens again. The `cm` directory stands for the cache manager, and contains cached advertisements. The files in that directory are also XML files; examining them is a good way to learn about the structure of JXTA advertisements. Finally, the `pse` directory contains credential information.

### **3.3.3.11.6 JXTA Prime Cruncher**

We will design and write a distributed JXTA application that solves parallel computing problems. We will construct this application in an iterative fashion, expanding its capabilities and the set of APIs it uses with each step.

A large subset of computational problems lend themselves to a parallel solution. Parallel execution of a task means that you break a problem into many smaller sub-problems, and cause those sub-problems to execute simultaneously. After a subtask completes, it returns its result to a master process, which then assembles the answer to the larger problem from those small results.

As an example, consider the task of creating a list of prime numbers between any two integers. Prime numbers are natural numbers that divide only by themselves and one. Natural numbers that divide by one and any other integer less than themselves are composite numbers. Thus, the simplest way to produce a list of prime numbers is to eliminate from a list of natural numbers all composites; the elements remaining in the list will all be primes.

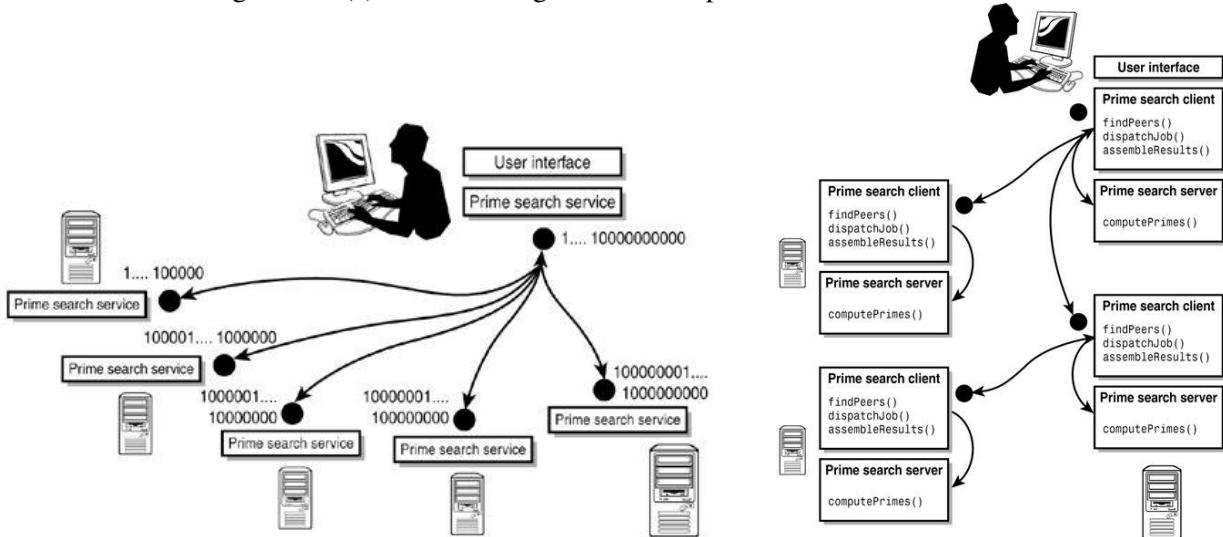
That method is the essence of a very old—albeit not very efficient—algorithm: the Sieve of Eratosthenes. It is named after Eratosthenes of Cyrene (ca. 275–195 B.C.), a mathematician chiefly known for being the first to accurately estimate the diameter of the Earth; he also served as director of the famous Alexandria library.

The Sieve of Eratosthenes identifies prime numbers by iterating through a list of natural numbers and attempting to eliminate from that list all composites. It does that by dividing every number in the list by each natural number between two and the square root of the ultimate number in the list. If any number in the list divides by a number other than itself without leaving a remainder, then that number is a composite and is marked as such. After the iterations complete, eliminating all marked numbers leaves only primes in the list. The following example illustrates how the Sieve works.

Consider the list of natural numbers between 10 and 20. To find all primes between these two numbers, we will divide every element in the list by 2, 3, and 4 (4.47 being the approximate square root of 20). The original list is as follows: 10, 11, 12, 13, 14, 15, 16, 17, 18, 19, and 20. Eliminating the numbers that divide by 2 without a remainder leaves 11, 13, 15, 17, and 19. Next, removing all numbers that divide by 3 leaves 11, 13, 17, and 19. Doing the divisions by 4 does not change the list. Therefore, we have obtained the complete list of primes between 10 and 20.

With a very long list of natural numbers—for instance, with numbers several million digits long—we can divide that list into multiple smaller lists, and perform the Sieve on each list simultaneously. Each of those computations might be handed out to different machines on the network, taking advantage of distributed computing resources. Prime number searching is but one of a large set of problems that can be parallelized. Among the popular uses of P2P-style software are applications such as the SETI@HOME project, which aims to decode signals from outer space in search of intelligent life on other planets, or similar projects enabling users to contribute their idle CPU resources to tasks such as simulating protein folding or decoding strands of DNA.

In this application, a master process will request two numbers from the user and produce a list of all primes between those two numbers. The master process will attempt to discover other peers on the JXTA network offering the prime number search service, and try to parcel out list segments to them for processing. After a peer completes its part of the work, it will send back an array of primes for its segment of the list. For that distribution to work, we will enable a JXTA peer to advertise its prime searching capability on the network so that others can find and connect to it. Figure 3.51(a) outlines this generic JXTA prime cruncher.



**Fig. 3.51. (a)** A server architecture suitable for finding large prime numbers in a distributed manner. **(b)** A peer offers both server-mode and client-mode operations (SM/CM).

## JXTA Application Design

Perhaps the most unusual aspect of this application is that every peer acts both as a master process and a slave, helping compute a sublist handed to it by a master. It is also conceivable that a slave might decide to further break down the problem into small subtasks, and act as a master process itself. This server-mode/client-mode operation is an essential P2P application design pattern. We will refer to that pattern as a SM/CM operation. It's worth noting that we will exploit SM/CM to reuse code: The master process itself will act as a slave to an adapter standing between it and the user interface: When a user specifies the two extremes of the natural number list, that adapter constructs the list and passes it to the prime cruncher component. Figure 3.51(b) illustrates this design.

**Message definition.** When designing a JXTA application, we must bear in mind that JXTA is a message-based system: The primary contract between peers is defined by a set of messages. Thus, the first design task is to define that message exchange. In the prime cruncher application, a peer passes a message to another peer containing the two boundaries of the list. The receiving peer then computes a list of all primes between those two extremes, and returns that sublist to the original peer. The net.jxta.endpoint.Message class abstracts out the concept of a message. It allows one to associate an arbitrary set of message elements with a key. We will use instances of that class with the following key-value structures seen in:

Key	Value
ServiceConstants.LOW_INT	Lower boundary of the (sub)list
ServiceConstants.HIGH_INT	Upper boundary of the (sub)list
ServiceConstants.PRIMELIST	A string containing all primes between the bounds of the list. The primes are separated by ; characters.

**Service Definition and Discovery.** Next we must define a way for a master to find slaves on the network. In other words, we must specify the prior knowledge a peer must have in order to discover other peers offering the prime crunching service. As mentioned earlier, a JXTA service is defined by its module class and specification. Thus, we will define advertisements for the number-crunching module's class and specification, and cause a peer offering that service to propagate those advertisements on the JXTA network. The prime-crunching module class will assume the name JXTACLASS:com.sams.p2p.primecruncher, and the module's spec will have the name JXTASPEC:com.sams.p2p.primecruncher. Masters will discover peers that advertise module specifications with that name. Thus, in addition to the message definition, the service name string is another piece of information peers must possess at design time. All other information pertinent to peer interaction will be discovered at runtime.

**Service Implementation.** When a prime-crunching peer starts up, it must first initialize the JXTA platform to gain access to the World and Net Peer Groups. The code for that initialization is similar to our earlier example. After the platform initiated, the peer creates and publishes its advertisements, including its module class and module spec advertisements.

The module spec advertisement will include the advertisement of a pipe. Clients discovering a module spec advertisement for the service must obtain the pipe advertisement, and connect to the service via that pipe.

After it has published its advertisements, our service opens an input pipe and listens for incoming messages. When a message arrives, the service attempts to obtain the high and low boundary numbers from it, and pass those onto a component responsible for generating the primes-only sublist. When that component returns its results (an array containing the primes), the prime cruncher service attempts to create a message with a result and then send that message back to the client. In the first iteration, the service will simply print out the message it receives. In subsequent refinements, it will open a pipe back to the client and send the results back to it. The client will then assemble the results from all the peers it heard back from and save the resulting master list into the file.

The outline of this server component is shown in what follows (Outline of PrimePeer and Initialization of a JXTA Peer):

```
package primecruncher;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.discovery.DiscoveryService;
import net.jxta.pipe.PipeService;
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.PipeID;
import net.jxta.exception.PeerGroupException;
import net.jxta.protocol.ModuleClassAdvertisement;
import net.jxta.protocol.ModuleSpecAdvertisement;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.document.*;
import net.jxta.platform.ModuleClassID;
import net.jxta.platform.ModuleSpecID;
import net.jxta.id.IDFactory;
import net.jxta.endpoint.Message;
```

```

import java.io.FileInputStream;
import java.io.IOException;
import java.io.FileOutputStream;
import java.io.StringWriter;
public class PrimePeer {
    private static PeerGroup group;
    private static DiscoveryService discoSvc;
    private static PipeService pipeSvc;
    private InputPipe inputPipe;
    private static final String PIPE_ADV_FILE = "primeserver_pipe.adv";
    public static void main(String[] argv) {
        PrimePeer pp = new PrimePeer();
        pp.startJxta();
        pp.doAdvertise();
        pp.startService();
    }
    public PrimePeer() {
    }
    private void startJxta() {
        try {
            group = PeerGroupFactory.newNetPeerGroup();
            discoSvc = group.getDiscoveryService();
            pipeSvc = group.getPipeService();
        } catch (PeerGroupException e) {
            System.out.println("Cannot create Net Peer Group: " + e.getMessage());
            System.exit(-1);
        }
    }
    /* Create and propagate advertisements*/
    private void doAdvertise() {
        ...
    }
    /*Start up the service,listen for incoming messages on service's input pipe.*/
    private void startService() {
        ...
    }
    /* Compute the requested list of prime numbers.*/
    private void processInput(String high, String low) {
        ...
    }
}

```

In the startJxta() service initialization method, we first obtain a reference to the World Peer Group; this is done via a static PeerGroupFactory method. Calling that method will cause the JXTA runtime to bootstrap. Next, we obtain references to two peer group services that the Net Peer Group provides: the DiscoveryService and the PipeService. We will use both when creating the service's advertisements.

**Creating and Publishing Advertisement.** As we mentioned earlier, the JXTA virtual network relies on JXTA IDs to identify network resources. The discovery of those resources occurs via advertisements. The net.jxta.id package contains the ID class, as well as a factory for creating various kinds of IDs—IDFactory. The code described below uses IDFactory to create a ModuleClassID for our new module.

In JXTA, a net.jxta.document.Document serves as a general container for data. A Document in JXTA is defined by the MIME media type of its content, and it has the capability of producing an InputStream with the content itself. In that sense, Document is somewhat analogous to an HTTP stream. JXTA makes no attempt to interpret the content of a Document; that content is part of an application-level protocol.

A Document that holds the advertisement of a JXTA network resource is a net.jxta.document.Advertisement. An Advertisement is a StructuredDocument, composed of a hierarchy of elements similar to XML. Structured documents can be nested, which enables a document to be manipulated without regard to the physical representation of its data.

As with any StructuredDocument, an Advertisement can be represented in XML or plain text formats. An Advertisement contains the ID of the resource it advertises, the type of the Advertisement, as well as an expiration time specified as an absolute time value. The JXTA API provides a convenient factory, AdvertisementFactory, to create different types of advertisements. The code described below shows the creation of a new ModuleClassAdvertisement via that factory class. Note the manner in which the ModuleClassID is added to the advertisement.

#### *Creating and Advertising a Module Class:*

```
private void doAdvertise() {
    ModuleClassAdvertisement classAd =
        (ModuleClassAdvertisement)AdvertisementFactory.newAdvertisement(
            ModuleClassAdvertisement.getAdvertisementType());
    ModuleClassID classID = IDFFactory.newModuleClassID();
    classAd.setModuleClassID(classID);
    classAd.setName(ServiceConstants.CLASS_NAME);
    classAd.setDescription("A prime number crunching service.");
    try {
        discoSvc.publish(classAd, DiscoveryService.ADV);
        discoSvc.remotePublish(classAd, DiscoveryService.ADV);
        System.out.println("Published module class adv.");
    } catch (IOException e) {
        System.out.println("Trouble publishing module class adv: " +
            e.getMessage());
    }
}
```

The JXTA net.jxta.discovery.DiscoveryService is a group service provided by the Net Peer Group, and its main purpose is to facilitate the publishing and discovery of advertisements. It provides two modes of both publishing and discovery: local and remote. The local mode has to do with the peer's local cache—local discovery means looking for advertisements in that cache, and local publishing means entering an advertisement into the local cache. Remote, as its name says, means performing discovery and publishing in the context of the entire peer group. Thus, query messages propagate throughout the JXTA virtual network in accord with the protocols we described previously, and responses are resolved to those queries as they arrive from the network. Thus, remote discovery is asynchronous—it might take quite a while for a desired advertisement type to be found on the JXTA network. The above described code shows both the remote and local publishing of the ModuleClassAdvertisement.

Similar to the preceding process, we create a ModuleSpec ID via the IDFFactory class, and its corresponding advertising is obtained from the AdvertisementFactory (see the following code).

#### *Creating a New ModuleSpecAdvertisement:*

```
ModuleSpecAdvertisement specAd =
    (ModuleSpecAdvertisement)AdvertisementFactory.newAdvertisement(
        ModuleSpecAdvertisement.getAdvertisementType());
ModuleSpecID specID = IDFFactory.newModuleSpecID(classID);
specAd.setModuleSpecID(specID);
specAd.setName(ServiceConstants.SPEC_NAME);
specAd.setDescription("Specification for a prime number crunching service");
specAd.setCreator("Sams Publishing");
specAd.setSpecURI("http://www.sampspublishing.com/p2p/primecruncher");
specAd.setVersion("Version 1.0");
```

Recall that a ModuleSpecAdvertisement defines a wire protocol, or a network behavior, to access a service. Thus, we need to provide a PipeAdvertisement as a parameter to the ModuleSpecAdvertisement. Because the module's advertisements will be cached by peers on the network, it is important to ensure that each ModuleSpecAdvertisement refers to the same pipe. Thus, we must save the pipe's advertisement to persistent storage and read that data from storage whenever creating a new pipe advertisement, as shown in the code below. (If the advertisement has not been saved to disk yet, create and save a new one.)

#### *Creating a Pipe Advertisement*

```
PipeAdvertisement pipeAd = null;
try {
    FileInputStream is = new FileInputStream(PIPE_ADV_FILE);
```

```

        pipeAd = (PipeAdvertisement)AdvertisementFactory. newAdvertisement(
            new MimeMediaType("text/xml"), is);
        is.close();
    } catch (IOException e) {
        pipeAd = (PipeAdvertisement)AdvertisementFactory. newAdvertisement(
            PipeAdvertisement.getAdvertisementType());
        PipeID pid = IDFFactory.newPipeID(group.getPeerGroupID());
        pipeAd.setPipeID(pid);
        //save pipeAd in file
        Document pipeAdDoc=pipeAd.getDocument(new MimeMediaType("text/xml"));
        try {
            FileOutputStream os = new FileOutputStream(PIPE_ADV_FILE);
            pipeAdDoc.sendToStream(os);
            os.flush();
            os.close();
            System.out.println("Wrote pipe advertisement to disk.");
        } catch (IOException ex) {
            System.out.println("Can't save pipe advertisement to file " +
                PIPE_ADV_FILE);
            System.exit(-1); } }

```

The following code segment saves a pipe advertisement to disk in XML format. For instance, one running of this code produced the following XML document:

```

<?xml version="1.0"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
<Id>
urn:jxta:uuid-59616261646162614E5047205032503382CCB236202640F5A242ACE15A8F9D7C04
</Id>
<Type> JxtaUnicast </Type>
</jxta:PipeAdvertisement>

```

We subsequently pass this new PipeAdvertisement as a parameter to the ModuleSpecAdvertisement:

```
specAd.setPipeAdvertisement(pipeAdv);
```

At this point, we are ready to publish the ModuleSpecAdvertisement both locally and remotely, as illustrated in:

```

try {
    discoSvc.publish(specAd, DiscoveryService.ADV);
    discoSvc.remotePublish(specAd, DiscoveryService.ADV);
    System.out.println("Published module spec adv");
} catch (IOException e) {
    System.out.println("Trouble publishing module spec adv: " +
        e.getMessage()); }

```

Finally, we create an InputPipe based on the pipe advertisement in:

```

//create an input pipe based on the advertisement
try {
    inputPipe = pipeSvc.createInputPipe(pipeAd);
    System.out.println("Created input pipe");
} catch (IOException e) {
    System.out.println("Can't create input pipe. " + e.getMessage());
}

```

These are all the steps needed to publish a new JXTA service. Recall that a module's class advertisement advertises the fact that the module functionality exists in a peer group; it is a fairly abstract concept, somewhat analogous to a Java interface that defines an API, but does not provide an implementation. A module's spec advertisement, on the other hand, specifies a wire protocol to access the service. In this case, that wire protocol consists of an InputPipe to which other peers can send messages. It is to that InputPipe that the messages specifying the two boundary numbers will arrive.

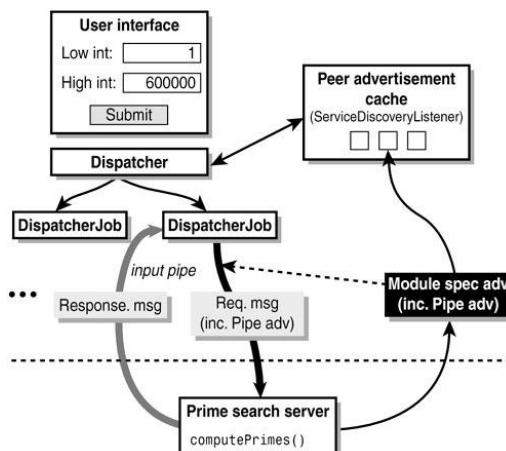
**Processing Messages from an InputPipe.** The next step in implementing the prime cruncher peer is to process the received messages. We will break this task down into handling incoming messages, calculating the desired list of prime numbers, and sending back a response. The following code shows the first part of the activity:

```
private void startService() {
    while (true) {
        Message msg = null;
        try {
            msg = inputPipe.waitForMessage();
        } catch (InterruptedException ex) {
            inputPipe.close();
            return;
        }
        String highInt = msg.getString(ServiceConstants.HIGH_INT);
        String lowInt = msg.getString(ServiceConstants.LOW_INT);
        if (highInt != null || lowInt != null) {
            processInput(highInt, lowInt);
        }
    }
}
```

As mentioned before, the `net.jxta.endpoint.Message` object is sent between two peers by `EndpointService` (an implementation of the Endpoint Protocol discussed earlier). A `Message` consists of a set of `MessageElements`, and features a destination `EndpointAddress` to facilitate its routing through the JXTA network. A message element can be any array of bytes, and `Message` has the capability to retrieve an element as a `String`. When a new message element is specified, it can be associated with a MIME type, as well as a `String` that serves as the element's key. In this method implementation, we retrieve the message elements referenced by the keys `ServiceConstants.HIGH_INT` and `ServiceConstants.LOW_INT`. If both elements are valid `Strings`, we pass them onto a private method, `processInput()`. `processInput()` is responsible for executing the Sieve of Eratosthenes algorithm (or any other algorithm) to produce a list of all prime numbers between `LOW_INT` and `HIGH_INT`. To save space, we will not show that part of the code here; instead, the full source code is available for download from [samspublishing.com](http://samspublishing.com). In addition, the full source code also contains a version of `startService()` that retrieves a `PipeAdvertisement` from the `Message` (as another message element), opens a pipe back to the client, and sends the list of prime numbers back to the client.

## The Prime Cruncher Client

The purpose of the client in this application is to distribute the computation load to as many peers advertising the number-crunching service as possible. Consider a user wanting to obtain all prime numbers between 1 and 10,000. When a peer receives that user request, it needs to determine how many other peers it can share the task with. Thus, it must continuously discover peers advertising the prime number service and maintain a cache of those peers' advertisements. If a peer has, for example, 10 other peers it can share the work with, then it might then create a message with `LOW_INT` set to 1 and `HIGH_INT` set to 1000, then another message with the numbers set to 1001 and 2000, respectively, and so forth. Finally, the client would open a pipe to each of the 10 peers, and transmit one message to each. Figure 3.52 describes that peer-to-peer message exchange.



*Fig. 3.52. The peer-to-peer message exchange.*

The client's skeleton looks similar to the server's. It also initializes the Net Peer Group, and obtains from it the group's discovery and pipe services:

```

package primecruncher;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.discovery.DiscoveryService;
import net.jxta.discovery.DiscoveryListener;
import net.jxta.discovery.DiscoveryEvent;
import net.jxta.pipe.PipeService;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.PipeID;
import net.jxta.exception.PeerGroupException;
import net.jxta.protocol.DiscoveryResponseMsg;
import net.jxta.protocol.ModuleSpecAdvertisement;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.MimeMediaType;
import net.jxta.document.TextElement;
import net.jxta.document.AdvertisementFactory;
import net.jxta.id.IDFactory;
import net.jxta.endpoint.Message;
import java.util.Enumeration;
import java.io.StringWriter;
import java.io.IOException;
import java.net.URL;
import java.net.MalformedURLException;
import java.net.UnknownServiceException;
import java.util.HashSet;
import java.util.Set;
public class PrimeClient implements DiscoveryListener {
    private static PeerGroup group;
    private static DiscoveryService discoSvc;
    private static PipeService pipeSvc;
    private OutputPipe outputPipe;
    private Set adverts = new HashSet();
    public PrimeClient() {}
    public static void main(String[] argv) {
        Client cl = new Client();
        cl.startJxta();
        cl.doDiscovery();
    }
    public int[] processPrimes(int low, int high) { }
    private void startJxta() {
        try {
            group = PeerGroupFactory.newNetPeerGroup();
            discoSvc = group.getDiscoveryService();
            pipeSvc = group.getPipeService();
        } catch (PeerGroupException e) {
            System.out.println("Can't create net peer group: " +
                e.getMessage());
            System.exit(-1);
        }
    }
    private void doDiscovery() { }
}

```

Although PrimePeer's key responsibility is to advertise its service and process incoming messages, PrimeClient must participate in the service discovery process. The doDiscovery() method initiates service discovery. First, the peer looks into its local cache for advertisements that match a Name attribute in the prime computing module's specification. It then processes each advertisement it finds there. Performing Local Discovery

```

System.out.println("Starting service discovery...");
System.out.println("Searching local cache for " +
    ServiceConstants.SPEC_NAME + " advertisements");

```

```

Enumeration res = null;
try {
    res = discoSvc.getLocalAdvertisements(DiscoveryService.ADV,
        "Name", ServiceConstants.SPEC_NAME);
} catch (IOException e) {
    System.out.println("IO Exception.");
}
if (res != null) {
    while (res.hasMoreElements()) {
        processAdv((ModuleSpecAdvertisement) res.nextElement());
    }
}

```

Next, the peer initiates remote advertisement discovery. Remote discovery means that discovery queries propagate through the JXTA network, and responses arrive as suitable advertisements are found. Thus, remote discovery is an asynchronous process. We pass a DiscoveryListener as an argument to DiscoveryService's getRemoteAdvertisements() method. In addition, we must also specify a threshold of the number of advertisements we desire to receive from each peer:

```

System.out.println("Starting remote discovery...");
discoSvc.getRemoteAdvertisements(null, DiscoveryService.ADV,
    "Name", ServiceConstants.SPEC_NAME, 1, this); }

```

Once remote discovery is initiated, discovered advertisements are cached in the local advertisement cache. So, the next time the peer starts up, it will likely discover advertisements from that cache.

DiscoveryListener specifies the discoveryEvent() method that gets called each time an advertisement matching our criteria is found. A DiscoveryEvent contains a DiscoveryResponseMsg, containing the actual advertisements found through remote discovery. We obtain an enumeration of those advertisements and process each, as seen in :

```

public void discoveryEvent(DiscoveryEvent event) {
    System.out.println("DiscoveryEvent called");
    DiscoveryResponseMsg mes = event.getResponse();
    //these contain the responses found
    Enumeration res = mes.getResponses();
    if (res != null) {
        while (res.hasMoreElements()) {
            processAdv((ModuleSpecAdvertisement) res.nextElement());
        }
    }
}

```

Our processAdv() method is very simple: It inserts each ModuleSpecAdvertisement into a set. A set ensures that no duplicate advertisements are stored. This set acts as a cache for module spec advertisements:

```

private void processAdv(ModuleSpecAdvertisement ad) {
    adverts.add(ad); }

```

**Advertisement Processing.** After we've set up a discovery listener, it will keep adding newly discovered module spec advertisements to our simple local cache. Each time the processPrimes() method gets called, the client peer will attempt to contact the peers represented by these module spec advertisements, connect to their input pipes, and pass a message that initiates the prime number search on each of those peers.

The first item in this method is to determine the set of peers we can delegate work to. Recall that an advertisement has an expiration date associated with it. Thus, we must eliminate advertisements that are no longer valid:

```

Public int[] processPrimes(int low, int high) {
    Set setCopy = null;
    synchronized(adverts) {
        Set setCopy = (Set) adverts.clone();
    }
    ArrayList workingList = new ArrayList();
    ArrayList expired = new ArrayList();

```

```

long currentTime = System.currentTimeMillis();
Iterator it = workingSet.iterator();
while (it.hasNext()) {
    ModuleSpecAdvertisement ad = (ModuleSpecAdvertisement)it.next();
    if (ad.getLocalExpirationTime() > currentTime + (2 * 60 *1000)) {
        workingList.addElement(ad);
    } else {
        expired.addElement(ad);
    }
}
removeExpired(expired);

```

The preceding code segment performs a simple cache management of discovered advertisements, delegating the removal of all advertisements that have either expired or about to expire shortly to the removeExpired() method (not shown here, but is included in the full source code).

After we have a set of valid advertisements, we can start processing them in order to obtain from them the pipe advertisement that we must use to send the messages. Because we assume (at least in this example) that all those advertisements refer to peers that we will actually use in our prime searching task, we first break down the job into smaller tasks corresponding to each peer.

Note that this job distribution is rather contrived: Some peers might be more capable than others, and some might have better network connections than others. Those differences should be taken into account when assigning tasks to a peer. Also, in practice it might not make sense to divide the job into too many small segments, because the network communication time could easily dominate the time spent on the actual processing of the prime numbers list. However, this example aims to illustrate how to obtain a pipe advertisement from a ModuleSpecAdvertisement, how to create a new message, and then how to send that message down the pipe.

The following code shows how the natural number list is broken into sublists, each sublist corresponding to a message that will be sent to a peer participating in the computation. Messages are then inserted into a hash map, and the key of the map indicates a message's status: Was it sent out already? Have we received a result for that computation yet?

### Creating New Messages

```

Map messageMap = new HashMap();
int size = workingList.size()
int mod = high % size;
high -= mod;
int perPiece = high / size;
for (int i=0; i < size; i++) {
    //create a new message
    Message msg = pipeSvc.createMessage();
    msg.setString(ServiceConstants.LOW_INT, low);
    //last message will get to compute a bit more
    if (i == size-1) {
        high = low + perPiece - 1 + mod;
    } else {
        high = low + perPiece -1;
    }
    msg.setString(ServiceConstants.HIGH_INT, high);
    low += perPiece;

    //we neither sent the message, nor did we get a response
    StatusMap statusMap = new StatusMap(false, false);
    StatusMap statusMap = new StatusMap(false, false);
    messageMap.put(statusMap, msg);
}

```

StatusMap is simply a pairing of two Boolean values; it is not listed here.

Our final step is to extract the pipe advertisements from each ModuleSpecAdvertisement, open each pipe, and send a message to that pipe. Finally, we will mark the message as sent.

Recall that an advertisement is just a structured document, similar to an XML document. It can easily be converted to a text document and printed out. It's useful to inspect the contents of the advertisement during development and at debug-time:

#### *Printing an Advertisement*

```

Collection ads = messageMap.values();
Iterator it = ads.iterator();
while (it.hasNext()) {
    ModuleSpecAdvertisement ad = (ModuleSpecAdvertisement)it.next();
    //First, print out ModuleSpec advertisement on standard output
    StructuredTextDocument doc =
        (StructuredTextDocument)ad.getDocument(
            new MimeMediaType ("text/plain"));
    try {
        StringWriter out = new StringWriter();
        doc.sendToWriter(out);
        System.out.println(out);
        out.close();
    } catch (IOException e) {
    }
}
...

```

As we discussed earlier, a StructuredTextDocument consists of elements, and one such element is a parameter. When we constructed the ModuleSpecAdvertisement for our service, we entered the service's pipe advertisement as a parameter. The parameter is just another StructuredDocument element that we can manipulate in the same way we would an XML document.

In parsing the advertisement's parameter element, we first obtain the pipe's ID and type. The pipe's ID conforms to a URN specification, outlined in the JXTA specifications, which encodes the 128-bit special identifier for the pipe. The following is an example of such a URN:

urn:jxta:uuid-59616261646162614E5047205032503382CCB236202640F5A242ACE15A8F9D7C04

The IDFactory class is capable of constructing the PipeID object from such a URN. That is the mechanism we use to assign the pipe ID to the pipe advertisement:

#### Working with Advertisement Parameters

```

StructuredTextDocument param = (StructuredTextDocument)ad.getParam();
String pipeID = null;
String pipeType = null;
Enumeration en = null;
if (param != null) {
    en = param.getChildren("jxta:PipeAdvertisement");
}
Enumeration child = null;
if (en != null) {
    child = ((TextElement)en.nextElement()).getChildren();
}
if (child != null) {
    while (child.hasMoreElements()) {
        TextElement el = (TextElement)child.nextElement();
        String elementName = el.getName();
        if (elementName.equals("Id")) {
            pipeID = el.getTextValue();
        }
        if (elementName.equals("Type")) {
            pipeType = el.getTextValue();
        }
    }
}
if (pipeID != null || pipeType != null) {
    PipeAdvertisement pipeAdvert = (PipeAdvertisement)
        AdvertisementFactory.newAdvertisement(

```

```

        PipeAdvertisement.getAdvertisementType());
try {
    URL pidURL = new URL(pipeID);
    PipeID pid = (PipeID)IDFactory.fromURL(pidURL);
    pipeAdvert.setPipeID(pid);
} catch (MalformedURLException e) {
    System.out.println("Wrong URL: " + e.getMessage());
    return;
} catch (UnknownServiceException e) {
    System.out.println("Unknown Service: " + e.getMessage());
    return;
}
}

```

Based on this PipeAdvertisement, we are now able to construct an output pipe that connects to the remote peer's input pipe, as shown in the following code. Recall that a pipe is unidirectional communication channel. Thus, we do not expect to hear back from the remote peer via this pipe. The remote peer performs an essentially similar task, opening a pipe back to the client and sending it a message with the results of the computation. (We do not show that part of the code here; please see the Web site for the full sample source code.)

### Creating an Output Pipe

```

try {
    outputPipe = pipeSvc.createOutputPipe(pipeAdvert, 30000);
    outputPipe.send(msg);
    System.out.println("Sent message on output pipe");
} catch (IOException e) {
    System.out.println("Can't send message through pipe: " + e.getMessage());
}
}

```

An interesting thing about this pipe-creation mechanism is that a peer might have changed network identities between sending out the ModuleSpecAdvertisement and a client contacting it for useful work. However, the peer's virtual identity on the JXTA network remains the same, and the runtime services ensure that the pipe advertised by the ModuleSpecAdvertisement connects.

After you have downloaded the full source code from the Web site and made the decision to run both peers on the same machine, you'll need to start them from separate directories and specify a different network communication port for each in the JXTA Configurator.

To start the server application, ensure that all the JXTA classes are in your classpath (this process is detailed in the JXTA installation document), and then type the following command:

```
java primecruncher.PrimePeer
```

You might also run the command with the following optional parameters. These parameters allow you to bypass the JXTA login screen:

```
java -Dnet.jxta.tls.principal=USERNAME
      -Dnet.jxta.tls.password=PASSWORD primecruncher.PrimePeer
```

By substituting your JXTA username and password you can run the client similarly:

```
java -Dnet.jxta.tls.principal=USERNAME
      -Dnet.jxta.tls.password=PASSWORD primecruncher.PrimeClient
```

The prime-finder application of this chapter operates as a full-fledged Java application, with its own user interface and main() method. In the next chapter, we will learn how to interactively invoke this application from the JXTA Shell.

### 3.3.11.7 P2P Web Services

We have seen Web services and P2P networks operating independently. We'll now discuss an important architectural model and use case scenarios in which Web services and P2P coexist to provide value-added applications.

#### SOAP-Over-P2P

The simplest logical model in which Web services can work over a P2P infrastructure is the operation of SOAP over P2P.

The largest number of SOAP deployments today work over HTTP. This means the XML payload of SOAP requests and responses travel over HTTP. To elaborate upon this idea, have a look at the following HTTP request that wraps and transports a SOAP (UDDI) request over the Internet (it's a HTTP request containing a SOAP request that in turn contains a UDDI request):

```
POST /services/uddi/testregistry/inquiryapi HTTP/1.1
Host: www-3.ibm.com
Content-type: text/xml; charset=utf-8
Content-length:509
SOAPAction: ""

<?xml version='1.0' encoding='UTF-8'?>
<SOAP-ENV:Envelope xmlns:xsi="http://www.w3.org/1999/XMLSchema-instance"
    xmlns:SOAP-ENV="http://schemas.xmlsoap.org/soap/envelope/"
    xmlns:xsd="http://www.w3.org/1999/XMLSchema"
        SOAP-ENV:encodingStyle="http://schemas.xmlsoap.org/soap/encoding">
<SOAP-ENV:Body>
    <find_business generic="1.0" xmlns="urn:uddi-org:api" maxRows="10">
        <findQualifiers />
        <name>%P2P%</name>
    </find_business>
</SOAP-ENV:Body>
</SOAP-ENV:Envelope>
```

The only addition to the normal XML payload in SOAP is the inclusion of HTTP headers.

If you want to operate SOAP-over-P2P, you'll transport the SOAP request's XML payload as such, without any headers. If you consider JXTA as an sample P2P network, the transport of SOAP messages will essentially be accomplished through JXTA pipes.

The JXTA peer is connected to the JXTA network through a pipe service. JXTA pipes are capable of receiving any type of data payload. For SOAP requests, this pipe will be used to carry XML payloads.

The SOAP client is responsible for interacting with SOAP servers. We can have two SOAP servers in: a local SOAP server, and a remote SOAP server.

The local server can host any applications that this peer would like to host for itself on its own machine. If this application did not have any JXTA module in it, it would be required to own an Internet address (most likely an HTTP URL such as <http://www.mySOAPService.com>), or a static IP address (like 216.23.4.6) to expose any of its Web services. But with this type of arrangement, it needs neither an HTTP URL nor a static IP address, and it can still host its SOAP services and expose them to the outside world.

If you don't believe what we're suggesting, try the application that we'll develop for this chapter. One of your friends can connect his computer to the Internet through normal dial-up (or any other way) and run the application. You can then access his SOAP services from your PC connected to the Internet!

This is exactly the purpose of P2P. Every peer is responsible to manage its own content (perhaps through SOAP services or by using any means it feels appropriate), and respond to queries from other peers. There is no longer the need to have central data repositories, search engines, and Web servers.

The remote SOAP server can be any SOAP server over the Internet. Our SOAP-over-P2P application can invoke this remote server through conventional client-server interaction. Other JXTA peers will not know whether the SOAP service being exposed resides locally or remotely.

**Use Cases for SOAP-Over-P2P.** We will now write two major use cases for our SOAP-over-P2P application. The two use cases involve advertising (or publishing) and service invocation.

The following actors will appear in this use case analysis:

- User—The user of our SOAP-over-P2P application.
- Publishing peer (or publisher)—An instance of our SOAP-over-P2P application that wants to advertise or publish a SOAP service over a JXTA network.
- Requesting peer (or requester)—An instance of our SOAP-over-P2P application that wants to invoke a SOAP service.
- SOAP server—The SOAP server on which there is a SOAP service deployed.
- Rendezvous point—A meeting point in a JXTA network where peers can meet. It is a JXTA service.

**Advertising Use Case.** While considering this use case, we are assuming that the service that we want to advertise over a JXTA network is already deployed on our local SOAP server. The deployment of services over a SOAP server is an independent process, and depends on which SOAP server you're using. Because our SOAP-over-P2P application is supposed to work with any SOAP server, we will not consider the deployment process in our use cases.

In this use case scenario, the user wants to advertise her SOAP services over a JXTA network. Through a GUI, she will ask the SOAP-over-P2P application to advertise the SOAP service on the JXTA network. The application will ask for the required data (name and description of the SOAP service). The user will provide the required data. The application will create an advertisement according to the data provided, and publish it at all known JXTA rendezvous points. Our SOAP-over-P2P application will also create an input pipe and start listening for service invocation requests.

**Service Invocation Use Case.** For this use case, we are assuming that you know the name of the SOAP service that you want to invoke. You do not need to know the location of the service implementations (the JXTA network can take care of this, as you'll shortly see), but you still need to know name of the service. The process of finding the name of the service might be part of a UDDI search process, and is not relevant in the present SOAP-over-P2P application.

Here, the user wants to invoke a SOAP service named P2PCarRentalService that resides somewhere across the JXTA network. He will ask our SOAP-over-P2P application to find the service. The application will author a search query to find a JXTA pipe service according to the name of the required SOAP service. The application (acting as a requesting peer) will send the search query to all known JXTA rendezvous points. The rendezvous points will match the pipe services published with them, and also forward the search request to other rendezvous points known to them. If and when the service is found, its advertisement is returned to the requesting peer.

Upon receipt of the pipe (SOAP) service advertisement, our SOAP-over-P2P application will open an output pipe and send the message invocation request to the peer who that's hosting the service. The requesting peer might receive more than one response. The reason for multiple responses is because there can be any number of publishers hosting the same SOAP service.

Our SOAP-over-P2P implementation will contain both advertising and requesting peers. This will enable the user to use this application for advertising its SOAP services and searching/invoking services from other JXTA peers.

## Classes in the SOAP-Over-P2P Application

Based on the preceding use case analysis, we can decide to implement the following classes in the SOAP-over-P2P application:

- JXTAPeer
- Publisher (derived from JXTAPeer)
- Requester (derived from JXTAPeer)

- JxtaGui (graphical user interface)

**JXTAPeer** is the main class that provides all JXTA functionality. We have designed the JXTAPeer class so that other JXTA-related classes will inherit from it for specific purposes. For example, Publisher will extend JXTAPeer and provide specific functionality to act as a publisher of SOAP services.

The JXTAPeer class is responsible for performing the following functions:

1. Make sure that all classes extending JXTAPeer will use the instance of JXTA. This is accomplished in the constructor, where we instantiate the static data member named group (of type PeerGroup). The data member group holds information about the peer group that we will join. While instantiating the group object, we will join the NetPeerGroup, which is the default peer group joined by all peers at startup.
2. Start JXTA. Starting the JXTA service is the responsibility of the startJXTA method, which is called from the constructor. The startJXTA method will instantiate three data members—groupAdvertisement (holds reference to the advertisement service of the peer group), disco (holds reference to the discovery service of the peer group), and pipes (holds reference to the pipe service of the peer group).
3. Create an input pipe to receive incoming messages and advertise it over the JXTA network. The PublishServiceOverJXTAPipe method of JXTAPeer class performs this function.
4. Create an output pipe and send outbound messages. The JXTAPeer class contains a method named CreateOutputPipeAndSendMessage that performs this function.

These four functions form the major interface of the JXTAPeer class. In addition, there are a few small functions that will be described after covering the details of the four methods.

We developed the JXTAPeer class by customizing different examples provided at JXTA.org. We have copied the licensing information in the JXTAPeer.java file.

We'll start with the constructor. Look at the following lines of code from the constructor:

```
public JXTAPeer() {
    if (group==null)
    {
        try {
            group = PeerGroupFactory.newNetPeerGroup();
        } catch (Exception e) {}
        objectCount++;
    }
    //if (!objectCount)
    startJXTA();
}
//constructor
```

This code makes sure that only one JXTA instance exists at a time. We have kept group as a static variable, so the newNetPeerGroup method will be called only once for all instances of the SOAP-over-P2P application.

The variable group is of type PeerGroup. Creating a new PeerGroup is the first step while instantiating JXTA. The current Java implementation of JXTA that we have used in building our SOAP-over-P2P application provides a class PeerGroupFactory, with static methods that can create new peer groups for us. We have used a static method (newNetPeerGroup) of this class to create a NetPeerGroup.

According to the JXTA specification, every peer at boot-time (while instantiating JXTA) joins the NetPeerGroup. Later on, a peer can join other groups as well, and there is no limitation on the number of groups a peer can join.

The next step is to get references to a few services for our NetPeerGroup. The following lines of code form the startJXTA method that's called from the constructor:

```
groupAdvertisement = group.getPeerGroupAdvertisement();
disco = group.getDiscoveryService();
pipes = group.getPipeService();
```

We will shortly require discovery and pipe services for this peer group, so we have called the `getDiscoveryService` and `getPipeService` methods of `PeerGroup` class. We will keep a reference to each of these services stored for future use.

The pipe service will be used whenever an input or outpipe pipe needs to be created. The discovery service is required while performing a search.

This finishes our constructor. We will now have a look at the `publishServiceOverJXTAPipe()` method of the `JXTAPeer` class.

**PublishServiceOverJXTAPipe()** method, as its name implies, will publish our SOAP service over the JXTA network. For this purpose, we will use a JXTA pipe service. A pipe service is a mechanism that enables JXTA applications to send and receive messages. If a JXTA application wants to send a message through a pipe service, it will create an output pipe on the service. Similarly, to receive messages, an application will create an input pipe on the pipe service. Think of a JXTA pipe service as a courier service, and the output and input pipes as the sending and receiving ends, respectively.

We will simply create a new pipe service, advertise it on `NetPeerGroup`, create an input pipe on the pipe service, and return a reference to the newly created pipe.

We will use this input pipe to represent our SOAP service. The name of the pipe will be the same as the name of our SOAP service. Other peer applications will search for our pipe service, and when found, create an output pipe to send service invocation messages.

Look at the `publishServiceOverJXTAPipe` method signature:

```
protected InputPipe publishServiceOverJXTAPipe(
    String ServiceName,
    String ServiceVersion,
    String ServiceDescription,
    String ServiceCreator,
    String SpecURI,
    String PipeAdvFile
) {
    try{ // Entire code for this method to be copied in this try block
    } catch (Exception ex) { return null; }
} // publishServiceOverJXTAPipe
```

This method takes in six parameters. We will use these parameters in creating the input pipe. `ServiceName` and `ServiceVersion` are the name and version of the SOAP service, respectively. `ServiceDescription` is a textual description of the service being published. `ServiceCreator` is the identification of a company or person advertising this service. `SpecURI` is a URI to a specification (descriptive) document that may reside anywhere (possibly over the Internet).

The last parameter, `PipeAdvFile`, is perhaps the most important. All JXTA services are specified by advertisements. Whether it is a pipe, discovery, or some other service, its details will be specified through XML-based advertisements. You can think about all JXTA services as logical entities that are supposed to function according to their respective advertisements. `PipeAdvFile` is the name of an XML file that we will use to advertise the JXTA pipe so that it can be used to exchange method invocation messages. We will shortly provide a sample of such an XML-based advertisement file.

The `PublishServiceOverJXTAPipe()` method follows three steps to publish a SOAP service as an input pipe. First, it creates a shortform pipe advertisement (an XML file that lets everyone know about the existence of a pipe) and publishes it over the JXTA network. This shortform advertisement only proves the existence of a service, and does not provide its details. In the current implementation of JXTA, the `ModuleClassAdvertisement` class provides the shortform advertisement. The following is the code to create and publish the shortform advertisement:

```
// Step 1: Copy this code in the beginning of try block.
ModuleClassAdvertisement mcadv = (ModuleClassAdvertisement) AdvertisementFactory.
    newAdvertisement(ModuleClassAdvertisement.getAdvertisementType());
```

```

mcadv.setName("JXTAMOD:"+ServiceName);
mcadv.setDescription(ServiceDescription);
ModuleClassID mcID = IDFactory.newModuleClassID();
mcadv.setModuleClassID(mcID);
disco.publish(mcadv, DiscoveryService.ADV);
disco.remotePublish(mcadv, DiscoveryService.ADV);

```

All advertisements are created from an AdvertisementFactory. After the AdvertisementFactory has created a ModuleClassAdvertisement object, you can call its set methods to specify its name, description, and ID (unique identification). Although the name and description come from the parameters passed to the publishServiceOverJXTAPipe() method, the ID needs to be created by another factory named IDFactory. After setting the name, description, and ID, you will use the DiscoveryService object created earlier in the constructor to publish this information on the local cache, as well as remotely on the NetPeerGroup.

The second step is to create a detailed advertisement for our pipe. This is handled by the ModuleSpecAdvertisement class. The purpose of using this advertisement in our application is to enable requesting peers to instantiate our advertisement in the form of a pipe service, so that they can send us method invocation requests through an output pipe. Have a look at the following lines of code that accomplish the creation and publishing of this detailed advertisement:

```

//Step 2, First segment: Copy this code after step 1.
//Create a ModuleSpecAdvertisement and
//call its set methods to specify parameters.
ModuleSpecAdvertisement mdadv = (ModuleSpecAdvertisement)AdvertisementFactory.
    newAdvertisement(ModuleSpecAdvertisement.getAdvertisementType());
mdadv.setName("JXTASPEC:"+ServiceName);
mdadv.setVersion(ServiceVersion);
mdadv.setCreator(ServiceCreator);
mdadv.setDescription(ServiceDescription);
mdadv.setModuleSpecID(IDFactory.newModuleSpecID(mcID));
mdadv.setSpecURI(SpecURI);

//Second segment
//Create a pipe advertisement by reading from an XML file.
//Then copy the pipe advertisement into
//the ModuleSpecAdvertisement created in the first segment.
PipeAdvertisement pipeadv = null;
try {
    FileInputStream is = new FileInputStream(PipeAdvFile);
    pipeadv = (PipeAdvertisement)
        AdvertisementFactory.newAdvertisement(new MimeMediaType("text/xml"), is);
    is.close();
} catch (Exception e) {
    receiver.setText(receiver.getText()+"\n"+ "Error reading advert file "+PipeAdvFile);
    return null;
}//catch

StructuredTextDocument paramDoc = (StructuredTextDocument)StructuredDocumentFactory.
newStructuredDocument (new MimeMediaType("text/xml"),"Parm");
StructuredDocumentUtils.copyElements(paramDoc, paramDoc,
    (Element)pipeadv.getDocument(new MimeMediaType("text/xml")));
mdadv.setParam((StructuredDocument) paramDoc);

//Third segment
//Publish (both locally and remotely) the ModuleSpecAdvertisement,
//which now contains our pipe advertisement.
disco.publish(mdadv, DiscoveryService.ADV);
disco.remotePublish(mdadv, DiscoveryService.ADV);

```

This code is divided into three segments. In the first segment, you will create a ModuleSpecAdvertisement and use its set methods in exactly the same way that you created a ModuleClassAdvertisement in Step 1.

In the second segment, you will create a pipe advertisement by reading from an XML file specified by PipeAdvFile. The XML file will provide an identification (Id), type, and the name of the pipe that we want to advertise.

We are going to create a JxtaUnicast-type pipe, which means it will be a unidirectional pipe that can have two end points (one transmitter and one receiver). The JXTA set of specifications also specifies a multicast type of pipe that can connect a single transmitter to many receivers.

The following is a simple XML pipe advertisement that can be used for this purpose:

```
<?xml version="1.0"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
    <Id>urn:jxta:uuid-9CCCDF5AD8154D3D87A391210404E59BE4B888209A2241A4A162A109
16074A9504
    </Id>
    <Type>JxtaUnicast</Type>
    <Name>JXTA-SOAP-SERVER</Name>
</jxta:PipeAdvertisement>
```

The root element in this advertisement is PipeAdvertisement, which belongs to the jxta namespace. It contains child elements to specify the ID, type, and name of the pipe being advertised.

After you have created a pipe advertisement, you will add it to the ModuleSpec-Advertisement that you created in the first segment. This means our pipe advertisement has to become part of the ModuleSpecAdvertisement. For this purpose, you will create a structured text document (a new empty StructuredTextDocument object), copy the pipe advertisement into it, then copy the StructuredTextDocument into the ModuleSpecAdvertisement.

The StructuredTextDocument class helps in the marshaling and demarshaling of XML or non-XML structured content. In our case, we want to load our pipe advertisement, which is XML data. Therefore, we have to specify the media type as text/XML. After the pipe advertisement has been copied into the ModuleSpecAdvertisement, our second segment is finished.

In the third segment of Step 2, you will simply publish the completed ModuleSpec-Advertisement, both in the local cache, as well as remotely on the JXTA network. You are now all set to create an input pipe, which will receive SOAP service method invocation requests. You will use the PipeService object named pipes that was created in the JXTAPeer constructor. Have a look at the following lines of code:

```
InputPipe ip = pipes.createInputPipe(pipeadv, ((listener==null)?this:listener));
return ip;
```

You will call the createInputPipe method of the pipes object and pass on two parameters:

- The pipe advertisement that we created in Step 2.
- A conditional statement to check whether there is a listener object available. The listener object is meant to listen to incoming messages and receive control upon the arrival of a new message.

We have a separate set method available to specify which object should act as a listener for our input pipe. The current JXTA implementation requires every listener to implement the PipeMsgListener interface. Our JXTAPeer class implements this interface (which is just one method, pipeMsgEvent, that will receive control upon the arrival of a new message) so it can act as a listener.

However if the application using our JXTAPeer wants to implement its own message receiving logic, it can call the setMessageReceiver method of the JXTAPeer class to specify the listener. If there is such a listener available, it will be used. If not, the JXTAPeer will itself act as the listener.

We will discuss later how to implement application-specific message listening logic.

**CreateOutputPipeAndSendMessage()**. This method creates an output pipe, and sends a message over it. Our SOAP-over-P2P application needs this functionality for two purposes:

- When a requesting peer wants to send a SOAP service invocation message to the publishing peer
- When a publishing peer wants to send a SOAP response message back to the requesting peer

You'll now see how an output pipe is created and a message sent over it. First look at the method signature:

```
public void createOutputPipeAndSendMessage(
    String ServiceName, //Name of output pipe service.
    String resBody//Message to be sent.)
{ //Method body ..... }
```

This method takes two parameters, namely ServiceName and resBody. ServiceName is the name of the input pipe service over which we want to create an output pipe (the input pipe will be the recipient of our message). We will first search for the input pipe advertisement on the JXTA network whose name matches with ServiceName:

```
OutputPipe myOutPipe;//Output pipe to send the message out.
Message msg = null;//The message to be sent.
Enumeration enum = null;
while (true) {
    try {
        enum = disco.getLocalAdvertisements(DiscoveryService.ADV, "Name", "JXTASPEC:"+ServiceName);
    if ((enum != null) && enum.hasMoreElements()) break;
    disco.getRemoteAdvertisements(null, DiscoveryService.ADV, "Name", "JXTASPEC:"+ServiceName, 1, null);
    Thread.sleep(2000);
    } catch (Exception e) { }
} //while
```

The preceding code works on a combination of two methods of the DiscoveryService class: getLocalAdvertisements and getRemoteAdvertisements. The GetLocal-Advertisements method will look for matching advertisements in the local cache. If there are none found, getRemoteAdvertisements will look for matching advertisements remotely on the JXTA network. If found, they will be loaded in the local cache, so that next call to getLocalAdvertisements can find and store them in an enumeration.

Your next task is to read the advertisement from the enumeration:

```
ModuleSpecAdvertisement pipeAdvertisement=(ModuleSpecAdvertisement) enum.nextElement();
```

We have read only the first element from the enumeration in order to keep the logic and this explanation simple.

You now have the advertisement stored in an object named moduleAdvertisement. Recall from the discussion of advertising a pipe service (Step 2 of PublishService-OverJXTAPipe method) that we published our pipe service as part of a detailed ModuleSpecAdvertisement. So what you have stored in the moduleAdvertisement object is actually the complete ModuleSpecAdvertisement, from which you will now extract the pipe advertisement. For this purpose, you will use a StructuredTextDocument (recall that we used the same class in copying a pipe advertisement into a module specification advertisement):

```
try {
    StructuredTextDocument paramDoc = (StructuredTextDocument) mdsadv.getParam();
    String pID = null;
    String pType = null;
    Enumeration elements = paramDoc.getChildren("jxta:PipeAdvertisement");
    elements = ((TextElement) elements.nextElement()).getChildren();
    while (elements.hasMoreElements()) {
        TextElement elem = (TextElement) elements.nextElement();
        String nm = elem.getName();
        if(nm.equals("Id")) {
            pID = elem.getTextValue();      continue;
        }
    }
}
```

```

} //if
if(nm.equals("Type")) {
    pType = elem.getTextValue();    continue;
} //if
}//while
//code blocks A and B should be copied here
} catch (Exception ex) {           }

```

The preceding code reads the Id and Type children of the jxta:PipeAdvertisement element. You will now form a pipe advertisement from the Id and Type values:

```

//code block A
PipeAdvertisement pipeadv =
(PipeAdvertisement) AdvertisementFactory.newAdvertisement(
    PipeAdvertisement.getAdvertisementType());
try {
    URL pipeID = new URL(pID );
    pipeadv.setPipeID( (PipeID) IDFFactory.fromURL( pipeID ) );
    pipeadv.setType(pType);
} catch ( MalformedURLException badID ) {           }

```

You are now all set to create an output pipe based on the pipe advertisement that we just formed, then author a message and send it on the output pipe:

```

//code block B
myOutPipe = pipes.createOutputPipe(pipeadv, 11000);
msg = pipes.createMessage();
msg.setString("ServiceName", ServiceName);
msg.setString("SOAPResponse", resBody);
myOutPipe.send (msg);

```

This finishes our discussion of creating an output pipe and sending a message over it. The JXTAPeer class also has three other small methods.

**SetInputPipeMessageListener** This method designates an object that will receive all messages destined for the input pipe that we created in the PublishServiceOverJXTAPipe() method:

```

public void setInputPipeMessageListener(PipeMsgListener msgListener) {
    listener = msgListener;
}

```

**PipeMsgEvent** This method is part of the PipeMsgListener interface that JXTAPeer implements in order to receive pipe messages. This method doesn't do anything, and is meant to be overridden in subclasses:

```

public void pipeMsgEvent ( PipeMsgEvent event ){ }

```

**SetMessageReceiver** This method specifies a JTextArea object that will display all messages (including the SOAP response message) on the GUI:

```

public void setMessageReceiver(JTextArea msgReceiver) {
    receiver = msgReceiver;
}//setMessageReceiver

```

We will now see how our Publisher and Requester classes will extend the JXTAPeer class. Most of the functionality required by the publishing and requesting processes is already covered in the JXTAPeer class. Therefore, these two classes only need to implement logic specific to publishing and requesting processes.

**Publisher.** This class is responsible for performing the following functions:

1. Publish a SOAP service over the JXTA network.
2. Listen for SOAP service invocation messages from requesting peers.

3. Invoke a local SOAP server upon receipt of a service invocation request.
4. Send the SOAP response from the SOAP server back to the requesting peer.

The first and last tasks are already implemented by the `publishServiceOverJXTAPipe` and `createOutputPipeAndSendMessage` methods of the `JXTAPeer` class. Application-level logic (for example, GUI classes) can directly call these methods to publish SOAP services and send responses.

We only need to take care of the middle two points in the `Publisher` class. The interface of the `Publisher` class is very simple. There are only two methods—a constructor and a method named `pipeMsgEvent`. The constructor relies entirely on the methods of the `JXTAPeer` (super) class, while the `pipeMsgEvent` method implements the middle two points.

**Publisher Constructor** simply calls the `JXTAPeer` constructor (super) and relies on it to check whether a JXTA instance already exists, and to start JXTA accordingly.

```
public Server() {
    super();
    setInputPipeMessageListener(this);
}//constructor
```

After calling the super's constructor, the `Publisher` constructor also sets itself as the listener for receiving all input pipe messages. This is important, because we need to implement comprehensive logic for SOAP service invocation requests. All of this is handled in the `pipeMsgEvent` method.

**PipeMsgEvent.** This method performs the following tasks:

1. Receive control whenever a SOAP service invocation message is detected over the input pipe.
2. Read the name of the service that is being invoked and author a SOAP request.
3. Send the SOAP request to a local SOAP server.
4. Receive the SOAP response from the SOAP server.
5. Read the name of another input pipe embedded within the SOAP service invocation message. The requesting peer is listening for a SOAP response message on this pipe.
6. Call `createOutputPipeAndSendMessage` to create an output pipe and send the SOAP response back to the requesting peer.

As usual, first have a look at the method signature:

```
public void pipeMsgEvent ( PipeMsgEvent event ){
    //method body
}
```

This method takes only one parameter `event`, an object of type `PipeMsgEvent`. The `event` object holds the complete message received from the requesting peer, so your first step should be to extract the message from the `event` object. You will call the `getMessage` method of the `PipeMsgEvent` class that returns the complete message:

```
Message SOAPRequest = event.getMessage();
```

Next you will read the name of the service being invoked from the received message:

```
String SOAPServiceName = SOAPRequest.getString("ServiceName");
```

Now it's time to author a SOAP request. You will author a very simple hard-coded SOAP request, in which the only dynamic part is the name of the service.

SOAP request authoring requires the creation of an XML file that contains a SOAP envelope, a SOAP body, and service-related elements. The following code authors the complete XML payload of our simple SOAP request:

```
// SOAP authoring:
StringBuffer sh = new StringBuffer();
```

```

sh.append("<?xml version='1.0' encoding='UTF-8'?>\r\n");
sh.append("<SOAP-ENV:Envelope");
sh.append("\r\n");
sh.append("    xmlns:xsi=\"http://www.w3.org/1999/XMLSchema-instance\"");
sh.append("\r\n");
sh.append("    xmlns:SOAP-ENV=\"http://schemas.xmlsoap.org/soap/envelope\"");
sh.append("\r\n");
sh.append("    xmlns:xsd=\"http://www.w3.org/1999/XMLSchema\"");
sh.append("\r\n");
sh.append("    SOAP-ENV:encodingStyle=\"http://schemas.xmlsoap.org/soap/encoding\"");
sh.append("\r\n");
sh.append("    <SOAP-ENV:Body>");
sh.append("\r\n");
sh.append("<NS:invoke xmlns:NS='"+SOAPServiceName+"'>");
sh.append("</NS:invoke>");
sh.append("    </SOAP-ENV:Body>\r\n");
sh.append("</SOAP-ENV:Envelope>\r\n");

```

The preceding code simply takes a `StringBuffer` and adds a hard-coded SOAP request to it. The only dynamic part is the name of the service that appears within the `SOAP-Env:Body` element.

You will now add HTTP headers to this SOAP request and send it to a local SOAP server. We have used a simple HTTP client implementation named `HttpConnection` for this purpose:

```

// HTTP related stuff.
HttpConnection connection;
String response = "";
try{
    connection = new HttpConnection("http://localhost/soap/servlet/rpcrouter", 8080,
        "text/xml", "POST", sh.toString());
    connection.setRequestProperty ("SOAPAction:", "\"\"");
    response = connection.call();
} catch(Exception e) {
    receiver.setText(receiver.getText()+"\n"+ "Error connecting to SOAP server");
}//catch

```

This is a very simple use of HTTP for SOAP transport between a SOAP client (our SOAP-over-P2P application) and a SOAP server.

The last step is to send the SOAP response back to the requesting peer. You will need to perform the following two steps for this purpose:

1. Get the name of the pipe on which the requesting peer is listening. This name is embedded inside the SOAP service invocation that the peer sent.
2. Call the `createOutputPipeAndSendMessage` and send the SOAP response on the output pipe.

This is accomplished by the following lines of code:

```

String ResponsePipeName = SOAPRequest.getString("SOAPResponse");
createOutputPipeAndSendMessage(ResponsePipeName, response);

```

**Requester** class is responsible for performing the following tasks:

1. Search for a given SOAP service on the JXTA network.
2. Create an output pipe and send a service invocation request to the publishing peer.
3. Display the SOAP response message to the user through the GUI classes.

The Requester class depends entirely on its parent `JXTAPeer` class for all its functions. GUI classes will create a `Requester` object, and call its methods to perform the first two of the preceding tasks. The third task is handled by the `pipeMsgEvent` method in the following simple lines of code, which are self-explanatory:

```

public void pipeMsgEvent ( PipeMsgEvent event ){
    receiver.setText(receiver.getText()+"Response received:\n");
    Message SOAPMessage = event.getMessage();
    receiver.setText(
        receiver.getText()+SOAPMessage.getString("SOAPResponse")+":\n");
}

```

**Graphical User Interface class (JxtaGui)** is a very simple class that contains the following:

- A set of radio buttons for choosing between publishing and requesting modes.
- Some text fields for data entry. Only two fields must be filled in: the name of the service and the name of the advertisement file. The name of the service is what you will publish as a publisher or search as a requester. As far as the name of the advertisement file is concerned, we have provided two sample advertisement files, publisher.adv and requester.adv, with the downloads for this chapter on the Web site.
- A button for publishing and requesting. Its caption changes according to the mode selected.
- A message window to display messages and progress.

The JxtaGui class has a main method that calls the class constructor. The constructor will instantiate the GUI components, manage their layout, and display them. The JxtaGui constructor also contains handlers for data entry events, such as typing in the text fields and pressing buttons. The event handler for the Publisher/Requester button is the most important one.

When the user presses the Publisher or Requester button (whose caption changes according to the mode selected), the event handler for the button is called:

```

button.addActionListener(new java.awt.event.ActionListener() {
    public void actionPerformed(ActionEvent e) {
        if (button.getText().equals("Publish")){
            publisher.publishServiceOverJXTAPipe(
                ServiceName, ServiceVersion, ServiceDescription, ServiceCreator,
                ServiceSpecURL, ServiceAdvertisementFileName
            );
        } else if (button.getText().equals("Invoke")){
            requester.publishServiceOverJXTAPipe(
                "requester", ServiceVersion, ServiceDescription,
                ServiceCreator, ServiceSpecURL, ServiceAdvertisementFileName
            );
            requester.createOutputPipeAndSendMessage(
                ServiceName, "requester");
        } } });

```

In the preceding code, you will first check whether the user wants to publish or request. If the user wants to publish, it calls the publishServiceOverJXTAPipe method of the publisher object and passes on the values read from the GUI.

If the user wants to send a SOAP service invocation request, the application needs to perform the following two tasks:

1. Publish an input pipe service named requester. Technically, this is the same type of input pipe that listens for SOAP service invocation requests, but the requester pipe will act as a return path for a response from the publishing peer.
2. Create an output pipe and send the SOAP service invocation message.

### **3.3.12 Web Services Using Spring**

Spring Framework is certainly one of the more popular and interesting products today available in the Java development area. It has definitely revolutionized the approach to designing software projects. With its aspect-oriented philosophy, and the introduction of the concept of "Inversion of Control (IoC)" or "Dependency Injection", it has made the process of building the architecture of a J2EE application easier and cleaner. But Spring is much more than an IoC container. It provides out-of-the-box patterns and templates, and integrates well with a number of other frameworks and tools, brings flexibility, modularity, and robustness.

This great community has made, among the others, its contribution to the web services area, and its name is "Spring Web Services" or "Spring-WS".

Therefore, an immediate advantage in adopting this framework is the inheritance of the Spring concepts and patterns, as well as, the re-use of the know-how you may have already consolidated. The loose coupling between the service contract (or interface) and its implementation is, for example, one of the first pros of this choice.

A mainstream web service recipe recommends starting the designing from the WSDL (contract-first) rather than from the Java code. Spring-WS pushes this approach further, driving the designer to start from the schemas (XSD) of the input and output XML messages. The WSDL will then be automatically generated from the XSDs.

The XML handling can be configured to use a DOM-based library (W3C DOM, JDOM, dom4j, XOM), SAX, StAX and XPath, while for XML binding, you can choose between JAXB, Castor, XMLBeans, JiBX, and XStream. Spring-WS has a powerful and flexible Message Dispatcher which can handle the XML message distribution. Particular care has been dedicated to the WS-Security aspects regarding Authentication, Digital signatures, Encryption, and Decryption.

Spring has good support for Remoting. The main Remoting protocols Spring supports are RMI, HTTP-based Remoting (using org.springframework.remoting.httpinvoker.HttpInvokerServiceExporter), Hessian, Burlap, Spring support for SOAP, and Spring-WS (Web Services).

#### **3.3.12.1 Overview of Spring-WS**

Spring-WS is available as a download different from core Spring from the site. Spring-WS support Contract-first style of WS development. Hence, the developers should be ready with the contract (WSDL) first to implement WS in Spring. This may not be trivial for every developer, especially for those who cannot create a WSDL by hand. The other alternative is to use some tools to author the contract, and then use Spring-WS for implementation.

We have already seen how to develop WS in Axis. Now, what we need more from Spring are its features such as:

- Dependency Injection
- Object Wiring

And for the reasons mentioned earlier, we also need a mechanism to generate or author the WSDL. Hence, to make the full process smooth and straightforward, we can use a mixed approach—using both Axis and Spring together so that we get best of both the worlds. We will see how to do that in this section.

Spring provides org.springframework.remoting.jaxrpc.ServletEndpointSupport, which is a convenience base class for JAX-RPC servlet endpoint implementations. It provides a reference to the current Spring application context, so that we can do bean lookup or resource loading.

#### **3.3.12.2 A Hello example**

We will use the Server side codebase we have used for Axis sample with slight variations. Hence, the code is repeated in this section.

**IHello** is a simple business interface, with a single method hello. Since we want to share this interface with clients too, we have placed this interface alone in a common folder that is ch03\03\_Spring\Common\src.

```
public interface IHello{
    String hello(String param);
}
```

Let us have an interface different from IHello to **IHelloWeb**, as our web service interface. So, we shall generate

our contract out of this interface only.

```
public interface IHelloWeb extends IHello{ }
```

Different from our Axis sample, **HelloWebService** here extends `ServletEndpointSupport`, so that we get a reference to the current Spring application context.

```
public class HelloWebService extends ServletEndpointSupport implements IHelloWeb{
    private IHello iHello;
    public HelloWebService(){
        System.out.println("Inside HelloWebService.HelloWebService..."); }
    protected void onInit() {
        System.out.println("Inside HelloWebService.onInit..."); this.iHello = (IHello) getWebApplicationContext().getBean("hello");
    }
    public void setHello(IHello iHello){
        this.iHello = iHello;
    }
    public String hello(String param){
        System.out.println("Inside HelloWebService.hello..."); return iHello.hello(param);
    }
}
```

Here in the `onInit` method, we get a reference to Spring context to resolve the bean with the name, `hello`. This bean refers to a different business bean, where we implement our business code which is explained next.

**Hello** is a spring bean, which we configure in the `applicationContext.xml`. This bean implements the business method.

```
public class Hello implements IHello{
    public Hello(){
        System.out.println("Inside Hello.Hello..."); }
    public String hello(String param){
        System.out.println("Inside Hello.hello..."); return "Hello " + param;
    }
}
```

The **applicationContext.xml** will have definitions of all Spring beans, and is placed in the folder `WebService\config`.

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="hello" class="com.binildas.apache.axis.AxisSpring.Hello">
    </bean>
</beans>
```

The **web.xml** placed in `WebService\config` will explain how we can hook the Spring context to the current web application context. When we package the web archive, we need to place the `applicationContext.xml` in the path specified in the `web.xml` (/WEB-INF/).

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app PUBLIC "-//Sun Microsystems, Inc./DTD Web
Application 2.3//EN" "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <listener>
        <listener-class>
            org.springframework.web.context.ContextLoaderListener
        </listener-class>
    </listener>

```

```

</listener-class>
</listener>
<context-param>
    <param-name>contextConfigLocation</param-name>
    <param-value>
        /WEB-INF/applicationContext.xml
    </param-value>
</context-param>
<servlet>
    <servlet-name>AxisServlet</servlet-name>
    <display-name>Apache-Axis Servlet</display-name>
    <servlet-class>
        org.apache.axis.transport.http.AxisServlet
    </servlet-class>
    <load-on-startup>1</load-on-startup>
</servlet>
<servlet-mapping>
    <servlet-name>AxisServlet</servlet-name>
    <url-pattern>/services/*</url-pattern>
</servlet-mapping>
</web-app>

```

Let us code the **client** too using Spring features, in a simple manner:

```

public class Client{
    private ApplicationContext ctx;
    private ClientObject clientObject;

    public Client(){
        String[] paths = {" applicationContextClient.xml"};
        ctx = new ClassPathXmlApplicationContext(paths);
        clientObject = (ClientObject) ctx.getBean("clientObject");
    }
    public void finalize()throws Throwable{
        super.finalize();
        clientObject = null;
        ctx = null;
    }
    private void test1(){
        log(clientObject.hello("Binil"));
    }
    public static void main(String[] args) throws Exception{
        Client client = new Client();
        client.test1();
    }
}

```

The Client makes use of another spring bean, ClientObject. We wire this bean in a second Spring configuration file, applicationContextClient.xml. The **ClientObject** is just a helper bean.

```

public class ClientObject{
    private IHello helloService;
    public void setHelloService(IHello helloService) {
        this.helloService = helloService;
    }
    public String hello(String param) {
        return helloService.hello(param);
    }
}

```

We inject a proxy to the remote web service into this bean. So, any calls can be delegated to the web service. The proxy configuration and wiring is done in applicationContextClient.xml.

In **applicationContextClient.xml**, we configure both the ClientObject bean and a proxy to the remote web

service. To configure the proxy, you define a JaxRpcPortProxyFactoryBean so that the proxy will implement the remote interface. As you have chosen Axis to implement your Spring-based web service, we will use Axis itself for the client side invocation too. So you must specify org.apache.axis.client.ServiceFactory as the service factory class to use. Then you also define other parameters for the JaxRpcPortProxyFactoryBean as shown in following code listing:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE beans PUBLIC "-//SPRING//DTD BEAN//EN"
"http://www.springframework.org/dtd/spring-beans.dtd">
<beans>
    <bean id="helloService"
class="org.springframework.remoting.jaxrpc.JaxRpcPortProxyFactoryBean">
        <property name="serviceFactoryClass">
            <value>org.apache.axis.client.ServiceFactory</value>
        </property>
        <property name="serviceInterface">
            value="com.binildas.apache.axis.AxisSpring.IHello"/>
        <property name="wsdlDocumentUrl">
            value="http://localhost:8080/AxisSpring/
                services/HelloWebService?wsdl"/>
        <property name="namespaceUri">
            value="http://AxisSpring.axis.apache.binildas.com"/>
        <property name="serviceName" value="IHelloWebService"/>
        <property name="portName" value="HelloWebService"/>
    </bean>
    <bean id="clientObject"
        class="com.binildas.apache.axis.AxisSpring.ClientObject">
        <property name="helloService" ref="helloService"/>
    </bean>
</beans>
```

The ant command will build both the server and the client codebase. At the end of the build, we will have the deployable web archive (AxisSpring.war). You can now transfer this archive to the webapps folder of your web server and restart your server. Assuming the deployment went fine, the WSDL for the web service will be available now at the URL <http://localhost:8080/AxisSpring/services/HelloWebService?wsdl>.

### **3.3.13 Web Services Using XFire**

XFire has been developed with the goal of obtaining better performance with respect to Axis 1.x (the de facto standard at that time). In fact, it uses a fast object model based on StAX.

It is simple and easy to use; it provides support for several binding libraries: JAXB, Castor, and XMLBeans. But the default is Aegis Binding—a fast binding mechanism with a small memory requirement.

XFire integrates well with many containers, among which are Spring and PicoContainer (another Inversion-of-Control framework). A set of easy-to-use client API makes it easy to build the client side, and to develop unit tests. The transports supported are HTTP, JMS, and Jabber/XMPP.

Recently, XFire and another web service framework, Celtix, have converged to a new product, **CXF 2.0**, which should be considered the continuation of XFire 1.x. The goal of this new tool is to go further in the directions of high performances, and ease of use. CXF supports a number of protocols in addition to SOAP, including REST (via Annotations) and CORBA. It adheres to several standards and WS-\* specifications. Its focus is also on being made embeddable into other programs and pushes on the code-first methodology instead of contract-first.

XFire is a new generation Java SOAP framework. XFire API is easy to use, and supports standards. Hence XFire makes SOA development much easier and straightforward. XFire is also highly performance oriented, since it is built on a low memory **StAX (Streaming API for XML)** model. Currently, XFire is available in version 2.0 under the name CXF.

**An example.** As usual, **IHello** is a simple Java business interface, defining a single method sayHello.

```
public interface IHello{
    String sayHello(String name); }
```

**HelloServiceImpl** is our web service implementation class, implementing IHello interface.

```
public class HelloServiceImpl implements IHello{
    private static long times = 0L;

    public HelloServiceImpl(){
        System.out.println("HelloServiceImpl.HelloServiceImpl()..."); 
    }
    public String sayHello(String name){
        System.out.println("HelloServiceImpl.sayHello (" + (++times) + ")");
        return "HelloServiceImpl.sayHello : HELLO! You just said:" + name;
    }
}
```

For XFire web services, we need to set up org.codehaus.xfire.transport.http.XFireConfigurableServlet as the Servlet. We then route all URL requests of pattern /services/ to XFireConfigurableServlet as shown in the **web.xml**.

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<!DOCTYPE web-app
    PUBLIC "-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
    "http://java.sun.com/dtd/web-app_2_3.dtd">
<web-app>
    <servlet>
        <servlet-name>XFireServlet</servlet-name>
        <display-name>XFire Servlet</display-name>
        <servlet-class>
            org.codehaus.xfire.transport.http.XFireConfigurableServlet
        </servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>XFireServlet</servlet-name>
        <url-pattern>/servlet/XFireServlet/*</url-pattern>
    </servlet-mapping>
    <servlet-mapping>
        <servlet-name>XFireServlet</servlet-name>
        <url-pattern>/services/*</url-pattern>
    </servlet-mapping>
</web-app>
```

The **services.xml** is the main XFire configuration file. Let us look into the sample file and understand it in detail.

```
<beans xmlns="http://xfire.codehaus.org/config/1.0">
    <service>
        <name>Hello</name>
        <namespace>http://xfire.binildas.com</namespace>
        <serviceClass>com.binildas.xfire.IHello</serviceClass>
        <implementationClass>
            com.binildas.xfire.HelloServiceImpl
        </implementationClass>
    </service>
</beans>
```

The name element is required and denotes the name of the service as exposed to the world. The optional namespace element specifies the target namespace for the service. The serviceClass denotes the name of the object you wish to make into a service, whereas the implementationClass denotes the implementation which you wish to use when the service is invoked.

The ant command will build both the server and the client codebase. At the end of the build, we will have the deployable web archive (HelloXFire.war). You can now transfer this archive to the webapps folder of your web server and restart your server. Assuming the deployment went fine, the WSDL for the web service will be available now at the URL <http://localhost:8080/HelloXFire/services/Hello?wsdl>.

### **3.3.14 Data and Services**

Having seen the basics of XML and XML-based services in the previous chapters, we are now ready to look into the big picture of enterprise landscape and see how all the pieces fit together. What is of interest for every enterprise user is information and every information starts from the basic building block, data. Data can reside in any data store, and can exist in many formats. Irrespective of that, you need to bring data to your table, do some massaging with your business use cases, and supply them as information. How do we do that in the SOA world, moving away from the traditional JDBC or **Object-relational mapping (OR mapping)** styles? And more interesting is, data can even exist in the form of services and if so, how do we combine multiple services just like we combine data from multiple JDBC query results?

#### **3.3.14.1 Java Data Objects (JDO)**

You all are perfectly comfortable with JDBC or few OR-mapping frameworks at least, like Hibernate or TopLink. Let us now look into a complementing standard of accessing data from your data store using a standard interface-based abstraction model of persistence in java that is, **Java Data Objects (JDO)**. The original JDO (JDO 1.0) specification is quite old and is based on **Java Specification Request 12 (JSR 12)**. The current major version of JDO (JDO 2.0) is based on JSR 243. The original specifications were done under the supervision of Sun and starting from 2.0, the development of the API and the reference implementation happens as an Apache open-source project.

We have been happily programming to retrieve data from relational stores using JDBC, and now the big question is do we need yet another standard, JDO? If you think that as software programmers you need to provide solutions to your business problems, it makes sense for you to start with the business use cases and then do a business analysis at the end of which you will come out with a **Business Domain Object Model (BDOM)**. The BDOM will drive the design of your entity classes, which are to be persisted to a suitable data store. Once you design your entity classes and their relationship, the next question is should you be writing code to create tables, and persist or query data from these tables (or data stores, if there are no tables). I would like to answer 'No' for this question, since the more code you write, the more are the chances of making errors, and further, developer time is costly. Moreover, today you may write JDBC for doing the above mentioned "technical functionalities", and tomorrow you may want to change all your JDBC to some other standard since you want to port your data from a relational store to a different persistence mechanism. To sum up, let us list down a few of the features of JDO which distinguishes itself from other similar frameworks.

- **Separation of Concerns:** Application developers can focus on the BDOM and leave the persistence details (storage and retrieval) to the JDO implementation.
- **API-based:** JDO is based on a java interface-based programming model. Hence all persistence behavior including most commonly used features of OR mapping is available as metadata, external to your BDOM source code. We can also Plug and Play (PnP) multiple JDO implementations, which know how to interact well with the underlying data store.
- **Data store portability:** Irrespective of whether the persistent store is a relational or object-based file, or just an XML DB or a flat file, JDO implementations can still support the code. Hence, JDO applications are independent of the underlying database.
- **Performance:** A specific JDO implementation knows how to interact better with its specific data store, which will improve performance as compared to developer written code.
- **J2EE integration:** JDO applications can take advantage of J2EE features like EJB and thus the enterprise features such as remote message processing, automatic distributed transaction coordination, security, and so on.

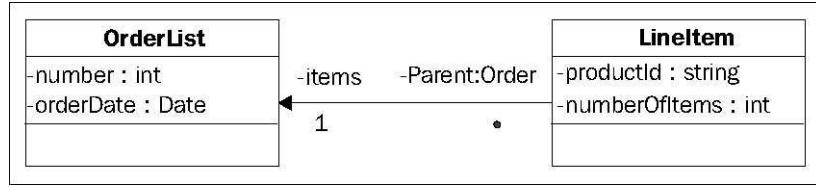
#### **3.3.14.2 JDO Using JPOX**

JPOX (Java Persistent Objects) is an Apache open-source project, which aims at a heterogeneous persistence solution for Java using JDO. By heterogeneous we mean, JPOX JDO will support any combination of the following four main aspects of persistence:

- **Persistence Definition:** The mechanism of defining how your BDOM classes are to be persisted to the data store.
- **Persistence API:** The programming API used to persist your BDOM objects.
- **Query Language:** The language used to find objects due to certain criteria.
- **Data store:** The underlying persistent store you are persisting your objects to.

We will take the familiar Order and LineItems scenario, and expand it to have a JDO implementation. It is assumed that you have already downloaded and extracted the JPOX libraries to your local hard drive.

We will limit our BDOM for the sample discussion to just two entity classes, that is, OrderList and LineItem. The class attributes and relationships are shown in the following screenshot.



**Figure 3.53 BDOM classes**

The BDOM illustrates that an Order can contain multiple line items. Conversely, each line item is related to one and only one Order.

The BDOM classes are simple entity classes with getter and setter methods for each attribute. These classes are then required to be wired for JDO persistence capability in a JDO specific configuration file, which is completely external to the core entity classes.

**OrderList** is the class representing the Order, and is having a primary key attribute that is number.

```

public class OrderList{
    private int number;
    private Date orderDate;
    private Set lineItems;

    // other getter & setter methods go here
    // Inner class for composite PK
    public static class Oid implements Serializable{
        public int number;

        public Oid(){      }
        public Oid(int param){ this.number = param;  }
        public String toString(){ return String.valueOf(number);   }
        public int hashCode(){ return number;      }
        public boolean equals(Object other){
            if (other != null && (other instanceof Oid)){
                Oid k = (Oid)other;
                return k.number == this.number;
            }
            return false;
        }
    }
}

```

**LineItem** represents each item container in the Order. We don't explicitly define a primary key for LineItem even though JDO will have its own mechanism to do that.

```

public class LineItem{
    private String productId;
    private int numberOfItems;
    private OrderList orderList;
    // other getter & setter methods go here
}

```

JDO requires an XML configuration file, which defines the fields that are to be persisted and to what JDBC or JDO wrapper constructs should be mapped to. For this, we can create an XML file called **package.jdo** with the following content and put it in the same directory where we have the entities.

```

<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jdo SYSTEM "file:/javax/jdo/jdo.dtd">
<jdo>
    <package name="com.binildas.jdo.jpox.order">
        <class name="OrderList" identity-type="application"
              objectid-class="OrderList$Oid" table="ORDERLIST">
            <field name="number" primary-key="true">
                <column name="ORDERLIST_ID"/>
            </field>
            <field name="orderDate">
                <column name="ORDER_DATE"/>
            </field>
            <field name="lineItems" persistence-modifier="persistent"
                  mapped-by="orderList">
                <collection element-type="LineItem">
                </collection>
            </field>
        </class>
        <class name="LineItem" table="LINEITEM">
            <field name="productId">
                <column name="PRODUCT_ID"/>
            </field>
            <field name="numberOfItems">
                <column name="NUMBER_OF_ITEMS"/>
            </field>
            <field name="orderList" persistence-modifier="persistent">
                <column name="LINEITEM_ORDERLIST_ID"/>
            </field>
        </class>
    </package>
</jdo>

```

In this sample, we will persist our entities to a relational database, Oracle. We specify the main connection parameters in **jpxo.PROPERTIES** file.

```

javax.jdo.PersistenceManagerFactoryClass=org.jpox.jdo.JDOPersistenceManagerFactory
javax.jdo.option.ConnectionDriverName=oracle.jdbc.driver.OracleDriver
javax.jdo.option.ConnectionURL=jdbc:oracle:thin:@127.0.0.1:1521:orcl
javax.jdo.option.ConnectionUserName=scott
javax.jdo.option.ConnectionPassword=tiger

org.jpox.autoCreateSchema=true
org.jpox.validateTables=false
org.jpox.validateConstraints=false

```

The **Main** class contains the code to test the JDO functionalities. As shown here, it creates two Orders and adds few line items to each order. First it persists these entities and then queries back these entities using the id.

```

public class Main{
    static public void main(String[] args) {
        Properties props = new Properties();
        try{
            props.load(new FileInputStream("jpox.properties"));
        }
        catch (Exception e){
            e.printStackTrace();
        }
        PersistenceManagerFactory pmf =
            JDOHelper.getPersistenceManagerFactory(props);
        PersistenceManager pm = pmf.getPersistenceManager();
        Transaction tx = pm.currentTransaction();
        Object id = null;

```

```

try{
    tx.begin();

    LineItem lineItem1 = new LineItem("CD011", 1);
    LineItem lineItem2 = new LineItem("CD022", 2);
    OrderList orderList = new OrderList(1, new Date());
    orderList.getLineItems().add(lineItem1);
    orderList.getLineItems().add(lineItem2);

    LineItem lineItem3 = new LineItem("CD033", 3);
    LineItem lineItem4 = new LineItem("CD044", 4);
    OrderList orderList2 = new OrderList(2, new Date());
    orderList2.getLineItems().add(lineItem3);
    orderList2.getLineItems().add(lineItem4);

    pm.makePersistent(orderList);
    id = pm.getObjectId(orderList);
    System.out.println("Persisted id : "+ id);

    pm.makePersistent(orderList2);
    id = pm.getObjectId(orderList2);
    System.out.println("Persisted id : "+ id);

    orderList = (OrderList) pm.getObjectById(id);
    System.out.println("Retrieved orderList : " + orderList);

    tx.commit();
}

catch (Exception e){
    e.printStackTrace();
    if (tx.isActive()){
        tx.rollback();
    }
}
finally{
    pm.close();
}
}
}
}

```

The above command will execute the following steps:

- First it compiles the java source files
- Then for every class you persist, use JPOX libraries to enhance the byte code.
- As the last step, we create the required schema in the data store.

### 3.3.14.3 Data Services

Good that you now know how to manage the basic data operations in a generic way using JDO and other techniques. By now, you also have good hands-on experience in defining and deploying web services. We all appreciate that web services are functionalities exposed in standard, platform, and technology neutral way. When we say functionality we mean the business use cases translated in the form of useful information. Information is always processed out of data. So, once we retrieve data, we need to process it to translate them into information.

When we define SOA strategies at an enterprise level, we deal with multiple **Line of Business (LOB)** systems; some of them will be dealing with the same kind of business entity. For example, a customer entity is required for a CRM system as well as for a sales or marketing system. This necessitates a **Common Data Model (CDM)**, which is often referred to as the Canonical Data Model or Information Model. In such a model, you will often have entities that represent "domain" concepts, for example, customer, account, address, order, and so on. So, multiple LOB systems will make use of these domain entities in different ways, seeking different information-based on the business context. OK, now we are in a position to introduce the next concept in SOA, which is "Data Services".

Data Services are specialization of web services which are data and information oriented. They need to manage the traditional **CRUD** (**C**reate, **R**ead, **U**pdate, and **D**elete) operations as well as a few other data functionalities such as search and information modeling. The Create operation will give you back a unique ID whereas Read, Update, and Delete operations are performed on a specific unique ID. Search will usually be done with some form of search criteria and information modeling, or retrieval happens when we pull useful information out of the CDM, for example, retrieving the address for a customer.

The next important thing is that no assumptions should be made that the data will be in a java resultset form or in a collection of transfer object form. Instead, you are now dealing with data in SOA context and it makes sense to visualize data in XML format. Hence, **XML Schema Definition (XSDs)** can be used to define the format of your requests and responses for each of these canonical data definitions. You may also want to use ad hoc queries using XQuery or XPath expressions, similar to SQL capabilities on relational data. In other words, your data retrieval and data recreation for information processing at your middle tier should support XML tools and mechanisms, and should also support the above six basic data operations. If so, higher level of abstractions in the processing tier can make use of the above data services to provide Application Specialization capabilities, specialized for the LOB systems. To make the concept clear, let us assume that we need to get the order status for a particular customer (`getCustomerOrderStatus()`) which will take the customer ID argument. The data services layer will have a retrieve operation passing the customer ID and the XQuery or the XPath statement will obtain the requested order information from the retrieved customer data. High level processing layers (such as LOB service tiers) can use high-level interface (for example, our `getCustomerOrderStatus` operation) of the Application Specialization using a web services (data services) interface and need not know or use XQuery or XPath directly. The underlying XQuery or XPath can be encapsulated, reused, and optimized.

### 3.3.14.4 Service Data Objects (SDO)

Data abstraction and unified data access are the two main concerns that any SOA-based architecture has to address. In the data services discussion, we talked a bit about data abstraction, by first defining data around domain entities and then decorating it with useful methods for data operations. Equally important is the issue of accessing heterogeneous data in a uniform way.

One of the main problems Service Data Objects (SDO) tries to solve is the issue of heterogeneous manner of data management. By data management, we mean data storage as well as operations on data lifecycle. SDO simplifies J2EE data programming model thus giving application developers more time to focus on the business problems.

SDO provides developers an API, the SDO API, and a programming model to access data. This API lets you to work with data from heterogeneous data sources, including RDBMS, entity EJBs, XML sources, web services, EIS data sources using the Java Connector Architecture, and so on. Hence you as a developer need not be familiar with a technology-specific API such as JDBC or XQuery in order to access and utilize data. Instead, you can just use SDO API.

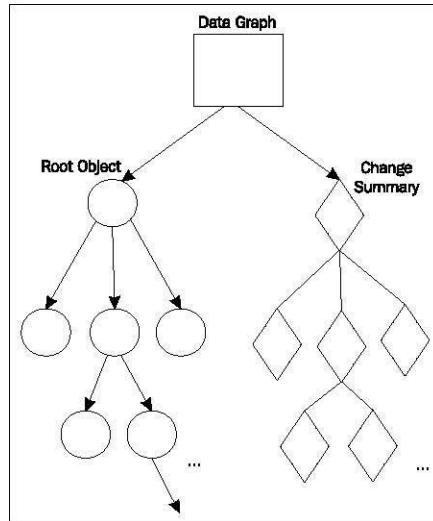
### 3.3.14.5 SDO Architecture

In SDO, data is organized as a graph of objects, called `DataObject`. A `DataObject` is the fundamental component which is a representation of some structured data, with some properties. These properties have either a single value or multiple values, and their values can be even other data objects. Each data objects also maintains a change summary, which represents the alterations made to it.

SDO clients or consumers always use SDO programming model and API. This is generic of technology and framework, and hence the developers need not know how the underlying data they are working with is persisted. A **Data Mediator Service (DMS)** is responsible for creating a data graph from data source(s), and also for updating the data source(s) based on any changes made to a data graph. SDO clients are disconnected from both the DMS and the data source.

A DMS will create a **Data Graph**, which is a container for a tree of data objects. Another interesting fact is that a single data graph can represent data from different data sources. This is actually a design model to deal with data aggregation scenarios from multiple data sources. The data graphs form the basics of the disconnected architecture of SDO, since they can be passed across layers and tiers in an application. When doing so, they are serialized to the XML format.

A **Change Summary** contains any change information related to the data in the data object. Change summaries are initially empty and are populated as and when the data graph is modified.



**Fig. 3.54 SDO**

### 3.3.14.6 Using Apache Tuscany SDO

Apache Tuscany SDO is a sub-project within open-source Apache Tuscany. Apache Tuscany aims at defining an infrastructure that simplifies the development of Service-Oriented application networks, addressing real business problems. It is based on specifications defined by the **OASIS Open Composite Services Architecture (CSA) Member Section**, which advances open standards that simplify SOA application development. Tuscany SDO mainly provides implementations in Java and C++. Both are available for download at: <http://incubator.apache.org/tuscany/>.

SDO can handle heterogeneous data sources, but for the sample here, we will make use of an XML file as a data source. The sample will read as well as write an XML file, when the client program makes use of SDO API to do data operations.

**An example.** The main artifacts for running the samples in SDO include an XSD schema file and an XML instance file. Then we have two java programs, one which reads the XML and another which creates an XML. We will look into these files first.

The **hr.xsd** restricts the structure of an employee XML file, which can contain multiple employees. Each employee can have a name, address, organization, and office elements. Each of these elements can have sub-elements, which are as shown here:

```

<?xml version="1.0"?>
<xsd:schema xmlns:xsd="http://www.w3.org/2001/XMLSchema"
  xmlns="http://www.binildas.com/apache/tuscany/sdo/sample"
  targetNamespace="http://www.binildas.com/apache/tuscany/sdo/sample">

  <xsd:element name="employees">
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element ref="employee" maxOccurs="unbounded" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
  <xsd:element name="employee">
    <xsd:annotation>
      <xsd:documentation>Employee representation</xsd:documentation>
    </xsd:annotation>
    <xsd:complexType>
      <xsd:sequence>
        <xsd:element name="name" type="xsd:string" />
        <xsd:element ref="address" maxOccurs="2" />
        <xsd:element ref="organization" />
        <xsd:element ref="office" />
      </xsd:sequence>
    </xsd:complexType>
  </xsd:element>
</xsd:schema>

```

```

        <xsd:attribute name="id" type="xsd:integer" />
    </xsd:complexType>
</xsd:element>
<xsd:element name="organization">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="name" type="xsd:string"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:integer" />
    </xsd:complexType>
</xsd:element>
<xsd:element name="office">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element ref="address"/>
        </xsd:sequence>
        <xsd:attribute name="id" type="xsd:integer" />
    </xsd:complexType>
</xsd:element>
<xsd:element name="address">
    <xsd:complexType>
        <xsd:sequence>
            <xsd:element name="street1" type="xsd:string"/>
            <xsd:element name="street2" type="xsd:string" minOccurs="0"/>
            <xsd:element name="city" type="xsd:string"/>
            <xsd:element name="state" type="stateAbbreviation"/>
            <xsd:element ref="zip-code"/>
        </xsd:sequence>
    </xsd:complexType>
</xsd:element>
<xsd:element name="zip-code">
    <xsd:simpleType>
        <xsd:restriction base="xsd:string">
            <xsd:pattern value="[0-9]{5}(-[0-9]{4})?"/>
        </xsd:restriction>
    </xsd:simpleType>
</xsd:element>
<xsd:simpleType name="stateAbbreviation">
    <xsd:restriction base="xsd:string">
        <xsd:pattern value="[A-Z]{2}"/>
    </xsd:restriction>
</xsd:simpleType>
</xsd:schema>

```

The **hr.xml** provided is fully constrained as per the above schema. For our sample demonstration this XML file contains data on two employees as shown here:

```

<?xml version="1.0"?>
<employees xmlns="http://www.binildas.com/apache/tuscany/sdo/sample">
    <employee id="30379">
        <name>Binildas C. A.</name>
        <address>
            <street1>45 Bains Compound Nanthencode</street1>
            <city>Trivandrum</city>
            <state>KL</state>
            <zip-code>695003</zip-code>
        </address>
        <organization id="08">
            <name>Software</name>
        </organization>
        <office id="31">
            <address>
                <street1>101 Camarino Ruiz</street1>
                <street2>Apt 2 Camarillo</street2>
                <city>Callifornia</city>
            </address>
        </office>
    </employee>
</employees>

```

```

        <state>LA</state>
        <zip-code>93012</zip-code>
    </address>
</office>
</employee>
<employee id="30380">
    <name>Rajesh R V</name>
    <address>
        <street1>1400 Salt Lake Road</street1>
        <street2>Appartment 5E</street2>
        <city>Boston</city>
        <state>MA</state>
        <zip-code>20967</zip-code>
    </address>
    <organization id="15">
        <name>Research</name>
    </organization>
    <office id="21">
        <address>
            <street1>2700 Cambridge Drive</street1>
            <city>Boston</city>
            <state>MA</state>
            <zip-code>20968</zip-code>
        </address>
    </office>
</employee>
</employees>

```

Now, we are going to see SDO in action. In the **ReadEmployees** class shown below, we first read the XML file, mentioned previously, and load it into a root DataObject. A DataObject is a graph of other DataObjects. Hence, we can iterate over the graph and get each item DataObject.

```

public class ReadEmployees extends SampleBase{

    private static final String HR_XML_RESOURCE = "hr.xml";
    public static final String HR_XSD_RESOURCE = "hr.xsd";

    public ReadEmployees(Integer commentaryLevel) {
        super(commentaryLevel,
              SampleInfrastructure.SAMPLE_LEVEL_BASIC);
    }

    public static void main(String[] args) throws Exception{
        ReadEmployees sample = new ReadEmployees(COMMENTARY_FOR_NOVICE);
        sample.runSample();
    }

    public void runSample () throws Exception{
        InputStream inputStream =
            ClassLoader.getSystemResourceAsStream(HR_XML_RESOURCE);
        byte[] bytes = new byte[inputStream.available()];
        inputStream.read(bytes);
        inputStream.close();

        HelperContext scope = createScopeForTypes();
        loadTypesFromXMLSchemaFile(scope, HR_XSD_RESOURCE);
        XMLDocument xmlDoc = getXMLDocumentFromString(scope,
            new String(bytes));
        DataObject purchaseOrder = xmlDoc.getRootObject();

        List itemList = purchaseOrder.getList("employee");
        DataObject item = null;
        for (int i = 0; i < itemList.size(); i++) {
            item = (DataObject) itemList.get(i);

```

```
        System.out.println("id: " + item.get("id"));
        System.out.println("name: " + item.get("name"));
    }
}
```

In the **CreateEmployees** class, we do the reverse process—we define DataObjects in code and build the SDO graph. At the end, the root DataObject is persisted to a file and also to the system output stream as shown in the following code.

```

public class CreateEmployees extends SampleBase {

    private static final String HR_XML_RESOURCE_NEW = "hr_new.xml";
    public static final String HR_XSD_RESOURCE = "hr.xsd";
    public static final String HR_NAMESPACE =
        "http://www.binildas.com/apache/tuscany/sdo/sample";

    public CreateEmployees(Integer commentaryLevel) {
        super(commentaryLevel, SAMPLE_LEVEL_BASIC);
    }
    public static void main(String[] args) throws Exception{
        CreateEmployees sample =
            new CreateEmployees(COMMENTARY_FOR_NOVICE);
        sample.runSample();
    }
    public void runSample() throws Exception{
        HelperContext scope = createScopeForTypes();
        loadTypesFromXMLSchemaFile(scope, HR_XSD_RESOURCE);
        DataFactory factory = scope.getDataFactory();
        DataObject purchaseOrder = factory.create(HR_NAMESPACE, "employees");

        DataObject employee1 = purchaseOrder.createDataObject("employee");
        employee1.setString("id", "3457");
        employee1.set("name", "Cindy Jones");
        DataObject homeAddress1 = employee1.createDataObject("address");
        homeAddress1.set("street1", "Cindy Jones");
        homeAddress1.set("city", "Stanchion");
        homeAddress1.set("state", "TX");
        homeAddress1.set("zip-code", "79021");
        DataObject organization1 = employee1.createDataObject("organization");
        organization1.setString("id", "78");
        organization1.set("name", "Sales");
        DataObject office1 = employee1.createDataObject("office");
        office1.setString("id", "43");
        DataObject officeAddress1 = office1.createDataObject("address");
        officeAddress1.set("street1", "567 Murdock");
        officeAddress1.set("street2", "Suite 543");
        officeAddress1.set("city", "Millford");
        officeAddress1.set("state", "TX");
        officeAddress1.set("zip-code", "79025");

        DataObject employee2 = purchaseOrder.createDataObject("employee");
        employee2.setString("id", "30376");
        employee2.set("name", "Linda Mendez");
        DataObject homeAddress2 = employee2.createDataObject("address");
        homeAddress2.set("street1", "423 Black Lake Road");
        homeAddress2.set("street2", "Appartment 7A");
        homeAddress2.set("city", "Boston");
        homeAddress2.set("state", "MA");
        homeAddress2.set("zip-code", "20967");
        DataObject organization2 = employee2.createDataObject("organization");
        organization2.setString("id", "78");
    }
}

```

```

organization2.set("name", "HR");
DataObject office2 = employee2.createDataObject("office");
office2.setString("id", "48");
DataObject officeAddress2 = office2.createDataObject("address");
officeAddress2.set("street1", "5666 Cambridge Drive");
officeAddress2.set("city", "Boston");
officeAddress2.set("state", "MA");
officeAddress2.set("zip-code", "20968");

OutputStream stream = new FileOutputStream(HR_XML_RESOURCE_NEW);
scope.getXMLHelper().save(purchaseOrder, HR_NAMESPACE,
    "employees", stream);
stream.close();

XMLDocument doc = scope.getXMLHelper().createDocument(purchaseOrder,
    HR_NAMESPACE, "employees");
scope.getXMLHelper().save(doc, System.out, null);
System.out.println();
}
}

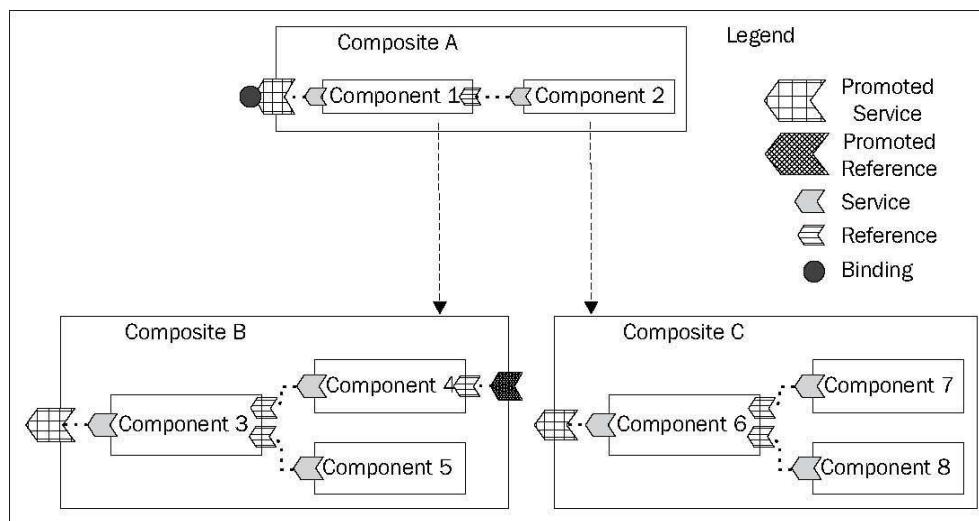
```

### 3.3.15 Service Component Architecture

We have been creating IT assets in the form of programs and codes since many years, and been implementing SOA architecture. This doesn't mean that we follow a big bang approach and throw away all old assets in place of new. Instead, the success of any SOA effort depends largely on how we can make the existing assets co-exist with new architecture principles and patterns. To this end, **Service Component Architecture (SCA)** aims at creating new and transforms existing, IT assets into re-usable services more easily. These IT assets can then be rapidly adapted to changing business requirements. In this section, we will introduce SCA and also look into some working samples for the same.

#### 3.3.15.1 SCA concept

SCA introduces the notion of services and references. A component which implements some business logic offers their capabilities through service-oriented interfaces. Components may also consume functionality offered by other components through service-oriented interfaces, called service references. If you follow SOA best practices, you will perhaps appreciate the importance of fine-grained tight coupling and coarse-grained loose coupling between components. SCA composition aids recursive assembly of coarse-grained components out of fine-grained tightly coupled components. These coarse-grained components can even be recursively assembled to form higher levels of coarse-grained components. In SCA, a composite is a recursive assembly of fine-grained components. All these are shown in the SCA assembly model in the Figure 3.55.



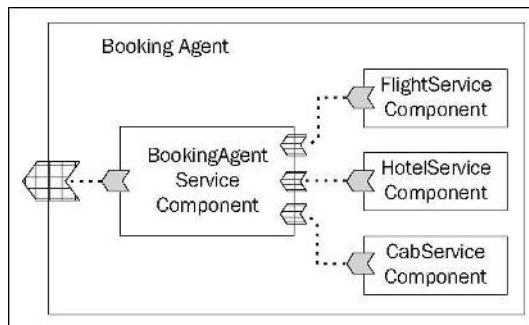
*Fig. 3.55 SCA*

## Using Apache Tuscany SCA Java

Apache Tuscany SCA is a sub-project within open-source Apache Tuscany, which has got a Java implementation of SCA. Tuscany SCA is integrated with Tomcat, Jetty, and Geronimo.

SCA Java runtime is composed of core and extensions. The core wires functional units together and provides SPIs that extensions can interact with. Extensions enhance SCA runtime functionality such as service discovery, reliability, support for transport protocols, and so on.

The sample here provides a single booking service with a default SCA (java) binding. The BookingAgentServiceComponent exercises this component by calling three other components that is, FlightServiceComponent, HotelServiceComponent, and CabServiceComponent as shown in the BookingAgent SCA assembly diagram shown below:



**Fig. 3.56 SCA Example**

The sample consists of two sets of artifacts. The first set is the individual fine-grained service components. The second set is the coarse-grained service component, which wires the referenced fine-grained service components.

There are three **fine-grained service components** whose code is self explanatory and are listed below:

```
FlightServiceComponent
public interface IFlightService{
    String bookFlight(String date, int seats, String flightClass);
}

public class FlightServiceImpl implements IFlightService{
    public String bookFlight(String date, int seats, String flightClass){
        System.out.println("FlightServiceImpl.bookFlight...");
        return "Success";
    }
}
```

```
HotelServiceComponent
public interface IHotelService{
    String bookHotel(String date, int beds, String hotelClass);
}

public class HotelServiceImpl implements IHotelService{
    public String bookHotel(String date, int beds, String hotelClass){
        System.out.println("HotelServiceImpl.bookHotel...");
        return "Success";
    }
}
```

```
CabServiceComponent
public interface ICabService{
    String bookCab(String date, String cabType);
}

public class CabServiceImpl implements ICabService{
    public String bookCab(String date, String cabType){
        System.out.println("CabServiceImpl.bookCab...");
        return "Success";
    }
}
```

```
}
```

**BookingAgentServiceComponent** depends on three referenced service components, which are the fine-grained service components listed previously. They are initialized by the dependency injection by the SCA runtime. Also, for the actual business method invocation, the call is delegated to the referenced service components as shown in the bookTourPackage method in the following code:

```
import org.osoa.sca.annotations.Reference;

public class BookingAgentServiceComponent implements IBookingAgent{
    private IFlightService flightService;
    private IHotelService hotelService;
    private ICabService cabService;

    @Reference
    public void setFlightService(IFlightService flightService) {
        this.flightService = flightService;
    }
    @Reference
    public void setHotelService(IHotelService hotelService) {
        this.hotelService = hotelService;
    }
    @Reference
    public void setCabService(ICabService cabService) {
        this.cabService = cabService;
    }
    public String bookTourPackage(String date, int people, String tourPack) {
        System.out.println("BookingAgent.bookTourPackage...");
        String flightBooked = flightService.bookFlight(date, people, tourPack);
        String hotelBooked = hotelService.bookHotel(date, people, tourPack);
        String cabBooked = cabService.bookCab(date, tourPack);

        if((flightBooked.equals("Success")) &&
           (hotelBooked.equals("Success")) &&
           (cabBooked.equals("Success"))){
            return "Success";
        }
        else{
            return "Failure";
        }
    }
}
```

The **BookingAgentClient** first creates an instance of SCADomain and then gets a reference of the BookingAgentServiceComponent using the name of the configured service component. Then it executes the business method, bookTourPackage.

```
import org.apache.tuscany.sca.host.embedded.SCADomain;

public class BookingAgentClient{
    public static void main(String[] args) throws Exception {
        SCADomain scaDomain = SCADomain.newInstance("BookingAgent.composite");
        IBookingAgent bookingAgent = scaDomain.getService(IBookingAgent.class,
            "BookingAgentServiceComponent");
        System.out.println("BookingAgentClient.bookingTourPackage...");
        String result = bookingAgent.bookTourPackage("20Dec2008", 5, "Economy");
        System.out.println("BookingAgentClient.bookedTourPackage : " + result);
        scaDomain.close();
    }
}
```

You can see that the BookingAgentServiceComponent will delegates calls to book individual line items to the referred service components and if all the individual bookings are done right, the overall transaction is "success".

## **3.3.16 WS-BPEL**

### **3.3.16.1 Orchestration**

Organizations that already have employed enterprise application integration (EAI) middleware products to automate business processes or to integrate various legacy environments will likely already be familiar with the concept of orchestration. In these systems, a centrally controlled set of workflow logic facilitates interoperability between two or more different applications. A common implementation of orchestration is the hub-and-spoke model that allows multiple external participants to interface with a central orchestration engine.

One of the driving requirements behind the creation of these solutions was to accommodate the merging of large business processes. With orchestration, different processes can be connected without having to redevelop the solutions that originally automated the processes individually. Orchestration bridges this gap by introducing new workflow logic. Further, the use of orchestration can significantly reduce the complexity of solution environments. Workflow logic is abstracted and more easily maintained than when embedded within individual solution components.

The role of orchestration broadens in service-oriented environments. Through the use of extensions that allow for business process logic to be expressed via services, orchestration can represent and express business logic in a standardized, services-based venue. When building service-oriented solutions, this provides an extremely attractive means of housing and controlling the logic representing the process being automated.

Orchestration further leverages the intrinsic interoperability sought by service designs by providing potential integration endpoints into processes. A key aspect to how orchestration is positioned within SOA is the fact that orchestrations themselves exist as services. Therefore, building upon orchestration logic standardizes process representation across an organization, while addressing the goal of enterprise federation and promoting service-orientation.

Web services can and eventually will be published for most software systems and applications within a given IT environment, and in fact across multiple organizations' IT environments. Rather than have Web services invoke each other using one or more of the message exchange patterns supported by SOAP and WSDL, an orchestration engine can be used to create more complex interaction patterns in long-running business process flows with exception handling, branching, and parallel execution. To accomplish this, the orchestration engine has to preserve context and provide correlation mechanisms across multiple services.

A Web service orchestration may also be published as a Web service, providing an interface that encapsulates a sequence of other Web services. Using the combination of MEPs and orchestration mechanisms, entire application suites can be built up out of Web services at multiple levels of encapsulation, from those that encapsulate a single software module to those that encapsulate a complex flow of other Web services.

The industry has reached a consensus around a single orchestration specification: the OASIS Web Services Business Process Execution Language (WS-BPEL). WS-BPEL assumes that Web services are defined using WSDL and policy assertions that identify any extended features.

Typically, a flow is initiated by the arrival of an XML document, and so the document-oriented Web services style tends to be used for modeling the entry point to a flow. Parts of the document are typically extracted and operated upon by the individual tasks in the flow, such as checking on the inventory availability for each line item from a different supplier, meaning the steps in the flow may be implemented using a combination of request/response and document-oriented Web services.

The WS-BPEL specification differs from other extended specifications in that it defines an executable language compatible with various software systems that drive business process automation. Whereas most other Web services specifications are XML representations of existing distributed computing features and capabilities that extend SOAP headers, orchestration represents the requirement for composing Web services in a declarative manner.

The workflow logic that comprises an orchestration can consist of numerous business rules, conditions, and events. Collectively, these parts of an orchestration establish a business protocol that defines how participants can interoperate to achieve the completion of a business task. The details of the workflow logic encapsulated and expressed by an orchestration are contained within a process definition.

Identified and described within a process definition are the allowable process participants. First, the process itself is represented as a service, resulting in a process service (which happens to be another one of our service models).

Other services allowed to interact with the process service are identified as partner services or partner links. Depending on the workflow logic, the process service can be invoked by an external partner service, or it can invoke other partner services.

WS-BPEL breaks down workflow logic into a series of predefined primitive activities. Basic activities (receive, invoke, reply, throw, wait) represent fundamental workflow actions which can be assembled using the logic supplied by structured activities (sequence, switch, while, flow, pick).

Basic and structured activities can be organized so that the order in which they execute is predefined. A sequence aligns groups of related activities into a list that determines a sequential execution order. Sequences are especially useful when one piece of application logic is dependent on the outcome of another.

Flows also contain groups of related activities, but they introduce different execution requirements. Pieces of application logic can execute concurrently within a flow, meaning that there is not necessarily a requirement for one set of activities to wait before another finishes. However, the flow itself does not finish until all encapsulated activities have completed processing. This ensures a form of synchronization among application logic residing in individual flows.

Links are used to establish formal dependencies between activities that are part of flows. Before an activity fully can complete, it must ensure that any requirements established in outgoing links first are met. Similarly, before any linked activity can begin, requirements contained within any incoming links first must be satisfied. Rules provided by links are also referred to as synchronization dependencies.

As we defined earlier, an activity is a generic term that can be applied to any logical unit of work completed by a service-oriented solution. The scope of a single orchestration, therefore, can be classified as a complex, and most likely, long-running activity.

Orchestration, as represented by WS-BPEL, can fully utilize the WS-Coordination context management framework by incorporating the WS-BusinessActivity coordination type. This specification defines coordination protocols designed to support complex, long-running activities.

Business process logic is at the root of automation solutions. Orchestration provides an automation model where process logic is centralized yet still extensible and composable. Through the use of orchestrations, service-oriented solution environments become inherently extensible and adaptive. Orchestrations themselves typically establish a common point of integration for other applications, which makes an implemented orchestration a key integration enabler.

These qualities lead to increased organizational agility because:

- The workflow logic encapsulated by an orchestration can be modified or extended in a central location.
- Positioning an orchestration centrally can significantly ease the merging of business processes by abstracting the glue that ties the corresponding automation solutions together.
- By establishing potentially large-scale service-oriented integration architectures, orchestration, on a fundamental level, can support the evolution of a diversely federated enterprise.

Orchestration is a key ingredient to achieving a state of federation within an organization that contains various applications based on disparate computing platforms. Advancements in middleware allow orchestration engines themselves to become fully integrated in service-oriented environments.

The concept of service-oriented orchestration fully leverages all of the concepts we've discussed so far in this chapter. For many environments, orchestrations become the heart of SOA.

The W3C's Web Services Choreography Definition Language (WS-CDL) is another specification in the general area of orchestration. Choreography is defined as establishing the formal relationship between two or more external trading partners. One difference between WS-CDL and WS-BPEL is that WS-CDL does not require Web services infrastructure at all of the endpoints being integrated.

### 3.3.16.2 Choreography

In a perfect world, all organizations would agree on how internal processes should be structured, so that should they ever have to interoperate, they would already have their automation solutions in perfect alignment.

Though this vision has about a zero percent chance of ever becoming reality, the requirement for organizations to interoperate via services is becoming increasingly real and increasingly complex. This is especially true when

interoperation requirements extend into the realm of collaboration, where multiple services from different organizations need to work together to achieve a common goal.

The Web Services Choreography Description Language (WS-CDL) is one of several specifications that attempts to organize information exchange between multiple organizations (or even multiple applications within organizations), with an emphasis on public collaboration. It is the specification we've chosen here to represent the concept of choreography and also the specification from which many of the terms discussed in this section have been derived.

An important characteristic of choreographies is that they are intended for public message exchanges. The goal is to establish a kind of organized collaboration between services representing different service entities, only no one entity (organization) necessarily controls the collaboration logic. Choreographies therefore provide the potential for establishing universal interoperability patterns for common inter-organization business tasks.

Within any given choreography, a Web service assumes one of a number of predefined roles. This establishes what the service does and what the service can do within the context of a particular business task. Roles can be bound to WSDL definitions, and those related are grouped accordingly, categorized as participants (services).

Every action that is mapped out within a choreography can be broken down into a series of message exchanges between two services. Each potential exchange between two roles in a choreography is therefore defined individually as a relationship. Every relationship consequently consists of exactly two roles.

Now that we've defined who can talk with each other, we require a means of establishing the nature of the conversation. Channels do exactly that by defining the characteristics of the message exchange between two specific roles.

Further, to facilitate more complex exchanges involving multiple participants, channel information can actually be passed around in a message. This allows one service to send another the information required for it to be communicated with by other services. This is a significant feature of the WS-CDL specification, as it fosters dynamic discovery and increases the number of potential participants within large-scale collaborative tasks.

Finally, the actual logic behind a message exchange is encapsulated within an interaction. Interactions are the fundamental building blocks of choreographies because the completion of an interaction represents actual progress within a choreography. Related to interactions are work units. These impose rules and constraints that must be adhered to for an interaction to successfully complete.

Each choreography can be designed in a reusable manner, allowing it to be applied to different business tasks comprised of the same fundamental actions. Further, using an import facility, a choreography can be assembled from independent modules. These modules can represent distinct sub-tasks and can be reused by numerous different parent choreographies

Finally, even though a choreography in effect composes a set of non-specific services to accomplish a task, choreographies themselves can be assembled into larger compositions.

While both represent complex message interchange patterns, there is a common distinction that separates the terms "orchestration" and "choreography." An orchestration expresses organization-specific business workflow. This means that an organization owns and controls the logic behind an orchestration, even if that logic involves interaction with external business partners. A choreography, on the other hand, is not necessarily owned by a single entity. It acts as a community interchange pattern used for collaborative purposes by services from different provider entities.

One can view an orchestration as a business-specific application of a choreography. This view is somewhat accurate, only it is muddled by the fact that some of the functionality provided by the corresponding specifications (WS-CDL and WS-BPEL) actually overlaps. This is a consequence of these specifications being developed in isolation and submitted to separate standards organizations (W3C and OASIS, respectively).

An orchestration is based on a model where the composition logic is executed and controlled in a centralized manner. A choreography typically assumes that there is no single owner of collaboration logic. However, one area of overlap between the current orchestration and choreography extensions is the fact that orchestrations can be designed to include multi-organization participants. An orchestration can therefore effectively establish cross-enterprise activities in a similar manner as a choreography. Again, though, a primary distinction is the fact that an orchestration is generally owned and operated by a single organization.

The fundamental concept of exposing business logic through autonomous services can be applied to just about any implementation scope. Two services within a single organization, each exposing a simple function, can interact via a basic MEP to complete a simple task. Two services belonging to different organizations, each exposing

functionality from entire enterprise business solutions, can interact via a basic choreography to complete a more complex task. Both scenarios involve two services, and both scenarios support SOA implementations.

Choreography therefore can assist in the realization of SOA across organization boundaries. While it natively supports composability, reusability, and extensibility, choreography also can increase organizational agility and discovery. Organizations are able to join into multiple online collaborations, which can dynamically extend or even alter related business processes that integrate with the choreographies. By being able to pass around channel information, participating services can make third-party organizations aware of other organizations with which they already have had contact.

### 3.3.16.3 WS-BPEL language basics

Before we can design an orchestration layer, we need to acquire a good understanding of how the operational characteristics of the process can be formally expressed. This book uses the WS-BPEL language to demonstrate how process logic can be described as part of a concrete definition that can be implemented and executed via a compliant orchestration engine.

Although you likely will be using a process modeling tool and will therefore not be required to author your process definition from scratch, a knowledge of WS-BPEL elements still is useful and often required. WS-BPEL modeling tools frequently make reference to these elements and constructs, and you may be required to dig into the source code they produce to make further refinements.

**A brief history of BPEL4WS and WS-BPEL.** Before we get into the details of the WS-BPEL language, let's briefly discuss how this specification came to be. The Business Process Execution Language for Web Services (BPEL4WS) was first conceived in July, 2002, with the release of the BPEL4WS 1.0 specification, a joint effort by IBM, Microsoft, and BEA. This document proposed an orchestration language inspired by previous variations, such as IBM's Web Services Flow Language (WSFL) and Microsoft's XLANG specification. Joined by other contributors from SAP and Siebel Systems, version 1.1 of the BPEL4WS specification was released in 2003. This version received more attention and vendor support, leading to a number of commercially available BPEL4WS-compliant orchestration engines. Later on has been announced that the language itself has been renamed to the Web Services Business Process Execution Language, or WS-BPEL (and assigned the 2.0 version number). Notes have been added to the element descriptions in this section where appropriate to indicate changes in syntax between BPEL4WS and WS-BPEL.

Let's begin with the root element of a WS-BPEL **process** definition. It is assigned a name value using the name attribute and is used to establish the process definition-related namespaces.

**Example 3.104. A skeleton process definition.**

```
<process name="TimesheetSubmissionProcess"
    targetNamespace="http://www.xmltc.com/tls/process/"
    xmlns=
        "http://schemas.xmlsoap.org/ws/2003/03/
         business-process/"
    xmlns:bpl="http://www.xmltc.com/tls/process/"
    xmlns:emp="http://www.xmltc.com/tls/employee/"
    xmlns:inv="http://www.xmltc.com/tls/invoice/"
    xmlns:tst="http://www.xmltc.com/tls/timesheet/"
    xmlns:not="http://www.xmltc.com/tls/notification/">
    <partnerLinks>
        ...
    </partnerLinks>
    <variables>
        ...
    </variables>
    <sequence>
        ...
    </sequence>
    ...
</process>
```

The process construct contains a series of common child elements explained in the following sections.

A **partnerLink** element establishes the port type of the service (partner) that will be participating during the execution of the business process. Partner services can act as a client to the process, responsible for invoking the process service. Alternatively, partner services can be invoked by the process service itself.

The contents of a partnerLink element represent the communication exchange between two partners—the process service being one partner and another service being the other. Depending on the nature of the communication, the role of the process service will vary. For instance, a process service that is invoked by an external service may act in the role of "TimesheetSubmissionProcess." However, when this same process service invokes a different service to have an invoice verified, it acts within a different role, perhaps "InvoiceClient." The partnerLink element therefore contains the myRole and partnerRole attributes that establish the service provider role of the process service and the partner service respectively.

Put simply, the myRole attribute is used when the process service is invoked by a partner client service, because in this situation the process service acts as the service provider. The partnerRole attribute identifies the partner service that the process service will be invoking (making the partner service the service provider).

Note that both myRole and partnerRole attributes can be used by the same partnerLink element when it is expected that the process service will act as both service requestor and service provider with the same partner service. For example, during asynchronous communication between the process and partner services, the myRole setting indicates the process service's role during the callback of the partner service.

**Example 3.105.** The partnerLinks construct containing one partnerLink element in which the process service is invoked by an external client partner and four partnerLink elements that identify partner services invoked by the process service.

```
<partnerLinks>
    <partnerLink name="client"
        partnerLinkType="tns:TimesheetSubmissionType"
        myRole="TimesheetSubmissionServiceProvider"/>
    <partnerLink name="Invoice"
        partnerLinkType="inv:InvoiceType" partnerRole="InvoiceServiceProvider"/>
    <partnerLink name="Timesheet"
        partnerLinkType="tst:TimesheetType"
        partnerRole="TimesheetServiceProvider"/>
    <partnerLink name="Employee"
        partnerLinkType="emp:EmployeeType" partnerRole="EmployeeServiceProvider"/>
    <partnerLink name="Notification"
        partnerLinkType="not:NotificationType"
        partnerRole="NotificationServiceProvider"/>
</partnerLinks>
```

Each of the partnerLink elements also contains a partnerLinkType attribute. This refers to the partnerLinkType construct, as explained next.

For each partner service involved in a process, **partnerLinkType** elements identify the WSDL portType elements referenced by the partnerLink elements within the process definition. Therefore, these constructs typically are embedded directly within the WSDL documents of every partner service (including the process service).

The partnerLinkType construct contains one role element for each role the service can play, as defined by the partnerLink myRole and partnerRole attributes. As a result, a partnerLinkType will have either one or two child role elements.

**Example 3.106.** A WSDL definitions construct containing a partnerLinkType construct.

```
<definitions name="Employee"
    targetNamespace="http://www.xmltc.com/tls/employee/wsdl/"
    xmlns="http://schemas.xmlsoap.org/wsdl/"
    xmlns:plnk= "http://schemas.xmlsoap.org/ws/2003/05/partner-link/"
    ...
    >
    ...
    <plnk:partnerLinkType name="EmployeeServiceType" xmlns=
        "http://schemas.xmlsoap.org/ws/2003/05/partner-link/">
        <plnk:role name="EmployeeServiceProvider">
            <portType name="emp:EmployeeInterface"/>
        </plnk:role>
    </plnk:partnerLinkType>
    ...

```

```
</definitions>
```

Note that multiple partnerLink elements can reference the same partnerLinkType. This is useful for when a process service has the same relationship with multiple partner services. All of the partner services can therefore use the same process service portType elements.

In version 2.0 of the WS-BPEL specification, it is being proposed that the portType element be changed so that it exists as an attribute of the role element.

WS-BPEL process services commonly use the **variables** construct to store state information related to the immediate workflow logic. Entire messages and data sets formatted as XSD schema types can be placed into a variable and retrieved later during the course of the process. The type of data that can be assigned to a variable element needs to be predefined using one of the following three attributes: messageType, element, or type.

The messageType attribute allows for the variable to contain an entire WSDL-defined message, whereas the element attribute simply refers to an XSD element construct. The type attribute can be used to just represent an XSD simpleType, such as string or integer.

**Example 3.107. The variables construct hosting only some of the child variable elements used later by the Timesheet Submission Process.**

```
<variables>
    <variable name="ClientSubmission"
        messageType="bpl:receiveSubmitMessage"/>
    <variable name="EmployeeHoursRequest"
        messageType="emp:getWeeklyHoursRequestMessage"/>
    <variable name="EmployeeHoursResponse"
        messageType="emp:getWeeklyHoursResponseMessage"/>
    <variable name="EmployeeHistoryRequest"
        messageType="emp:updateHistoryRequestMessage"/>
    <variable name="EmployeeHistoryResponse"
        messageType="emp:updateHistoryResponseMessage"/>
    ...
</variables>
```

Typically, a variable with the messageType attribute is defined for each input and output message processed by the process definition. The value of this attribute is the message name from the partner process definition.

WS-BPEL provides built-in functions that allow information stored in or associated with variables to be processed during the execution of a business process.

- **getVariableProperty**(variable name, property name) allows global property values to be retrieved from variables. It simply accepts the variable and property names as input and returns the requested value.
- **getVariableData**(variable name, part name, location path) Because variables commonly are used to manage state information, this function is required to provide other parts of the process logic access to this data. The getVariableData function has a mandatory variable name parameter and two optional arguments that can be used to specify a part of the variable data.

In our examples we use the getVariableData function a number of times to retrieve message data from variables.

**Example 3.108. Two getVariableData functions being used to retrieve specific pieces of data from different variables.**

```
getVariableData ('InvoiceHoursResponse', 'ResponseParameter')
getVariableData ('input','payload', '/tns:TimesheetType/Hours/...')
```

The **sequence** construct allows you to organize a series of activities so that they are executed in a predefined, sequential order. WS-BPEL provides numerous activities that can be used to express the workflow logic within the process definition. The remaining element descriptions in this section explain the fundamental set of activities used as part of our upcoming case study examples.

**Example 3.109. A skeleton sequence construct containing only some of the many activity elements provided by WS-BPEL.**

```
<sequence>
    <receive> ... </receive>
    <assign> ... </assign>
    <invoke> ... </invoke>
    <reply> ... </reply>
```

```
</sequence>
```

Note that sequence elements can be nested, allowing you to define sequences within sequences.

The **invoke** element identifies the operation of a partner service that the process definition intends to invoke during the course of its execution. The invoke element is equipped with five common attributes, which further specify the details of the invocation (Table 3.19).

**Table 3.19 invoke element attributes**

Attribute	Description
partnerLink	This element names the partner service via its corresponding partnerLink.
portType	The element used to identify the portType element of the partner service.
operation	The partner service operation to which the process service will need to send its request.
inputVariable	The input message that will be used to communicate with the partner service operation. Note that it is referred to as a variable because it is referencing a WS-BPEL variable element with a messageType attribute.
outputVariable	This element is used when communication is based on the request-response MEP. The return value is stored in a separate variable element.

**Example 3.110. The invoke element identifying the target partner service details.**

```
<invoke name="ValidateWeeklyHours"
    partnerLink="Employee"
    portType="emp:EmployeeInterface"
    operation="GetWeeklyHoursLimit"
    inputVariable="EmployeeHoursRequest"
    outputVariable="EmployeeHoursResponse"/>
```

The **receive** element allows us to establish the information a process service expects upon receiving a request from an external client partner service. In this case, the process service is viewed as a service provider waiting to be invoked.

The receive element contains a set of attributes, each of which is assigned a value relating to the expected incoming communication (Table 3.20).

**Table 3.20. Receive element attributes**

Attribute	Description
partnerLink	The client partner service identified in the corresponding partnerLink construct.
portType	The process service portType that will be waiting to receive the request message from the partner service.
operation	The process service operation that will be receiving the request.
variable	The process definition variable construct in which the incoming request message will be stored.
createInstance	When this attribute is set to "yes," the receipt of this particular request may be responsible for creating a new instance of the process.

Note that this element also can be used to receive callback messages during an asynchronous message exchange.

**Example 3.111. The receive element used in the Timesheet Submission Process definition to indicate the client partner service responsible for launching the process with the submission of a timesheet document.**

```
<receive name="receiveInput"
    partnerLink="client"
    portType="tns:TimesheetSubmissionInterface"
    operation="Submit"
    variable="ClientSubmission"
    createInstance="yes"/>
```

Where there's a receive element, there's a **reply** element when a synchronous exchange is being mapped out. The reply element is responsible for establishing the details of returning a response message to the requesting client partner service. Because this element is associated with the same partnerLink element as its corresponding receive element, it repeats a number of the same attributes (Table 3.21).

**Table 3.21. reply element attributes**

Attribute	Description
partnerLink	The same partnerLink element established in the receive element.
portType	The same portType element displayed in the receive element.
operation	The same operation element from the receive element.
variable	The process service variable element that holds the message that is returned to the partner service.
messageExchange	It is being proposed that this optional attribute be added by the WS-BPEL 2.0 specification. It allows for the reply element to be explicitly associated with a message activity capable of receiving a message (such as the receive element).

**Example 3.112. A potential companion reply element to the previously displayed receive element.**

```
<reply partnerLink="client"
      portType="tns:TimesheetSubmissionInterface"
      operation="Submit"
      variable="TimesheetSubmissionResponse"/>
```

These three structured activity elements allow us to add conditional logic to our process definition, similar to the familiar select case/case else constructs used in traditional programming languages. The **switch** element establishes the scope of the conditional logic, wherein multiple **case** constructs can be nested to check for various conditions using a condition attribute. When a condition attribute resolves to "true," the activities defined within the corresponding case construct are executed.

The **otherwise** element can be added as a catch all at the end of the switch construct. Should all preceding case conditions fail, the activities within the otherwise construct are executed.

**Example 3.113. A skeleton case element wherein the condition attribute uses the getVariableData function to compare the content of the EmployeeResponseMessage variable to a zero value.**

```
<switch>
  <case condition= "getVariableData('EmployeeResponseMessage',
    'ResponseParameter')=0">
    ...
  </case>
  <otherwise>
    ...
  </otherwise>
</switch>
```

It has been proposed that the switch, case, and otherwise elements be replaced with if, elseif, and else elements in WS-BPEL 2.0.

The set of elements {**assign**, **copy**, **from**, **to**} simply gives us the ability to copy values between process variables, which allows us to pass around data throughout a process as information is received and modified during the process execution.

**Example 3.114. Within this assign construct, the contents of the TimesheetSubmissionFailedMessage variable are copied to two different message variables.**

```
<assign>
  <copy>
    <from variable="TimesheetSubmissionFailedMessage"/>
    <to variable="EmployeeNotificationMessage"/>
  </copy>
  <copy>
    <from variable="TimesheetSubmissionFailedMessage"/>
    <to variable="ManagerNotificationMessage"/>
  </copy>
</assign>
```

Note that the copy construct can process a variety of data transfer functions (for example, only a part of a message can be extracted and copied into a variable). From and to elements also can contain optional part and query attributes that allow for specific parts or values of the variable to be referenced.

The construct **faultHandlers**, **catch** and **catchall** can contain multiple catch elements, each of which provides activities that perform exception handling for a specific type of error condition. Faults can be generated by the receipt of a WSDL-defined fault message, or they can be explicitly triggered through the use of the throw element. The faultHandlers construct can consist of (or end with) a catchAll element to house default error handling activities.

**Example 3.115. The faultHandlers construct hosting catch and catchAll child constructs.**

```
<faultHandlers>
    <catch faultName="SomethingBadHappened" faultVariable="TimesheetFault">
        ...
    </catch>
    <catchAll>
        ...
    </catchAll>
</faultHandlers>
```

## Other WS-BPEL elements

The following table provides brief descriptions of other relevant parts of the WS-BPEL language.

**Table 3.22. Quick reference table providing short descriptions for additional WS-BPEL elements (listed in alphabetical order).**

Element	Description
compensationHandler	A WS-BPEL process definition can define a compensation process that kicks in a series of activities when certain conditions occur to justify a compensation. These activities are kept in the compensationHandler construct.
correlationSets	WS-BPEL uses this element to implement correlation, primarily to associate messages with process instances. A message can belong to multiple correlationSets. Further, message properties can be defined within WSDL documents.
empty	This simple element allows you to state that no activity should occur for a particular condition.
eventHandlers	The eventHandlers element enables a process to respond to events during the execution of process logic. This construct can contain onMessage and onAlarm child elements that trigger process activity upon the arrival of specific types of messages (after a predefined period of time, or at a specific date and time, respectively).
exit	See the terminate element description that follows.
flow	A flow construct allows you to define a series of activities that can occur concurrently and are required to complete after all have finished executing. Dependencies between activities within a flow construct are defined using the child link element.
pick	Similar to the eventHandlers element, this construct also can contain child onMessage and onAlarm elements but is used more to respond to external events for which process execution is suspended.
scope	Portions of logic within a process definition can be sub-divided into scopes using this construct. This allows you to define variables, faultHandlers, correlationSets, compensationHandler, and eventHandlers elements local to the scope.
terminate	This element effectively destroys the process instance. The WS-BPEL 2.0 specification proposes that this element be renamed exit.
throw	WS-BPEL supports numerous fault conditions. Using the tHRow element allows you to explicitly trigger a fault state in response to a specific condition.
wait	The wait element can be set to introduce an intentional delay within the process. Its value can be a set time or a predefined date.
while	This useful element allows you to define a loop. As with the case element, it contains a condition attribute that, as long as it continues resolving to "true," will continue to execute the activities within the while construct.

### **3.3.17 Web Services with Eclipse WTP**

In this chapter Web services will be developed in iterations. In Iteration 1 you develop a Web service using the Top-Down approach. This means you create the description of the Web service interface first using the XSD and WSDL editors, and then generate its Java skeleton using the Web service wizard. You fill in the implementation of the Web service by writing Java code that accesses the. Finally, you test the Web service using the Web Services Explorer. In Iteration 2 you develop a Web service using the Bottom-Up approach. This means you write Java, and then use the Web service wizard to deploy the Java code as a Web service and generate its WSDL. In Iteration 3 you create a Web client that uses the update Web service. You use the Web service wizard to generate a Java client proxy from the WSDL of the Web service and a JSP test client that uses the proxy. In Iteration 4 you test your Web service for interoperability. You use the TCP/IP monitor and the WS-I test tools to test your Web service for compatibility with the WS-I profiles. In Iteration 5 you use your Web services in a Web application that displays schedules and updates game scores. The Web application accesses the Web services using Java client proxies. In Iteration 6 you use the Web Services Explorer to discover Web services in UDDI and WSIL registries. You also use the Web service wizard to publish WSIL documents that describe your Web services.

#### **3.3.17.1 Iteration 1: Developing Web Services Top-Down**

*Top-Down development* means designing the Web service interface first and then developing the implementation code. This approach yields the best interoperability because the underlying implementation details cannot “bleed through” into the interface. Top-Down development is required if the messages must use existing industry or corporate standard XML document formats. To perform Top-Down development you need to have XSD and WSDL design skills. Luckily, WTP has two great editors that make this task easier.

In this iteration, you'll perform the following tasks:

- 1 Use the XSD editor to describe the specific format of the data to be transmitted.
- 2 Use the WSDL editor to describe the Web service.
- 3 Use the Web service wizard to generate a Java skeleton for the service and deploy it to the Axis SOAP engine running on Tomcat.
- 4 Fill in the implementation of the Java skeleton by accessing the service logic tier.
- 5 Use the Web Services Explorer to test the schedule query service.

**XML Schema Description (XSD)** is the W3C Recommendation for describing the format or *schema* of XML documents, and is the preferred schema description language for use with Web services. XSD is far more expressive than its predecessor, DTD, and, like many specifications produced by industrial collaborations, is extremely feature rich. Fortunately, only a small portion of the XSD language is needed in practice to describe typical Web service messages.

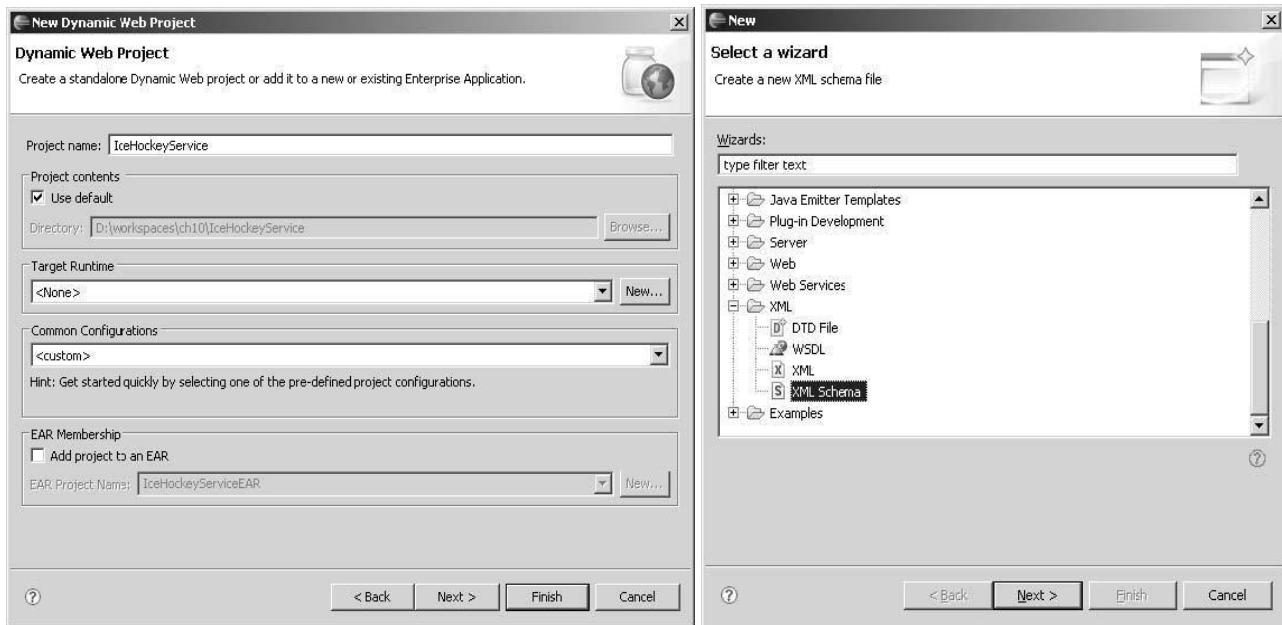
WTP has a powerful XSD editor that includes both a source and a graphical view as well as an outline view and property sheets that greatly simplify the editing task.

To create the schema for the service, do the following:

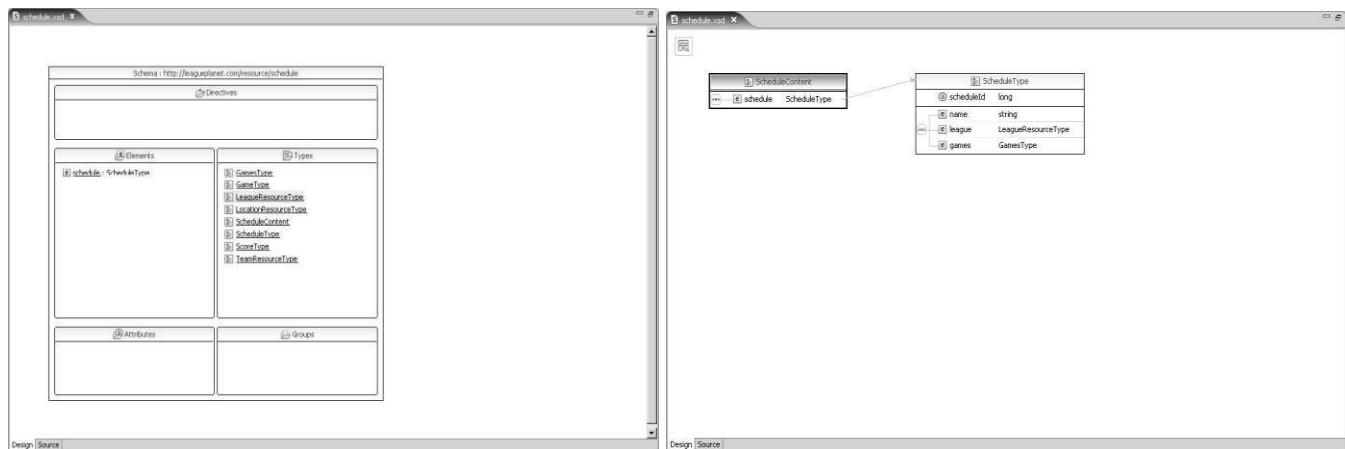
1. Create a new dynamic Web project to contain the Web service (see Figure 3.57(a)).
2. Import the XML with the data format.
3. Create a new XML Schema file with the extension xsd (see Figure 3.57(b)).

In general, there are many equivalent ways to describe a given format using XSD. For Web services, it's a good practice to describe formats in a way that works well with XML data binding toolkits such as JAX-RPC and JAX-WS. Define complex types for the content model of each element. The XSD editor lets you edit in the source tab, the graphical tab, the out-line view, and the property view. Try to develop the xsd file yourself.

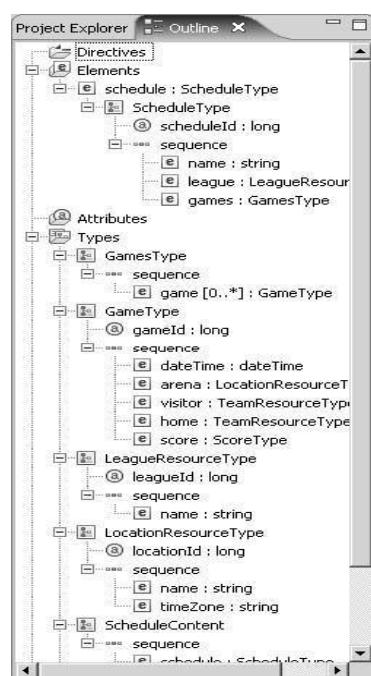
The XSD editor provides two types of graphical views. The first is an overview of the entire schema. This view acts like a visual table of contents. It arranges the definitions in the schema into the main top-level categories such as global element declarations and type definitions. A view of the xsd in the Graph tab of the XSD editor is seen in Figure 3.58 (a). The second type of graphical view is the detailed structure of an element declaration or type definition. View a complex type definition in the Graph tab of the XSD editor (see Figure 3.58(b)).



**Fig. 3.57. Screenshots of Eclipse interface - 1**



**Fig. 3.58. Screenshots of Eclipse interface - 2**

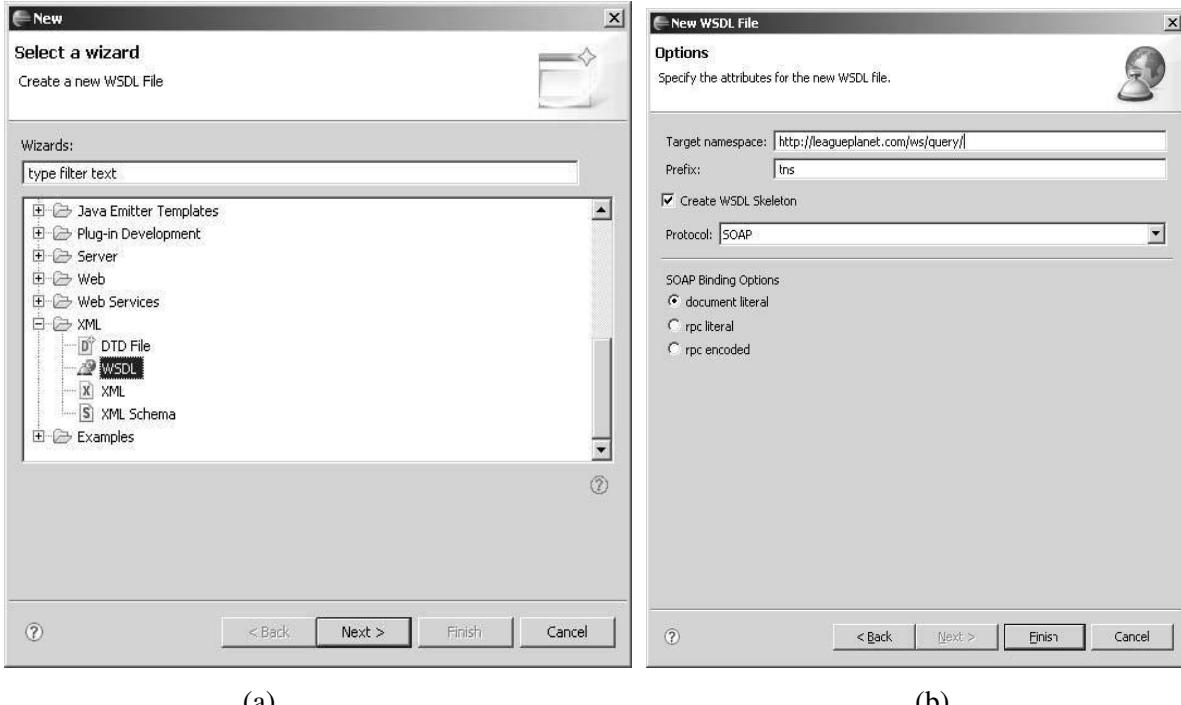


**Fig. 3.59. Screenshots of Eclipse interface - 3**

The XSD editor is linked to an outline view. You can edit the schema from this view. View the xsd in the Outline view of XSD editor (see Figure 3.59).

Now that you've described the message format using XSD, your next goal is to describe a Web service for retrieving it.

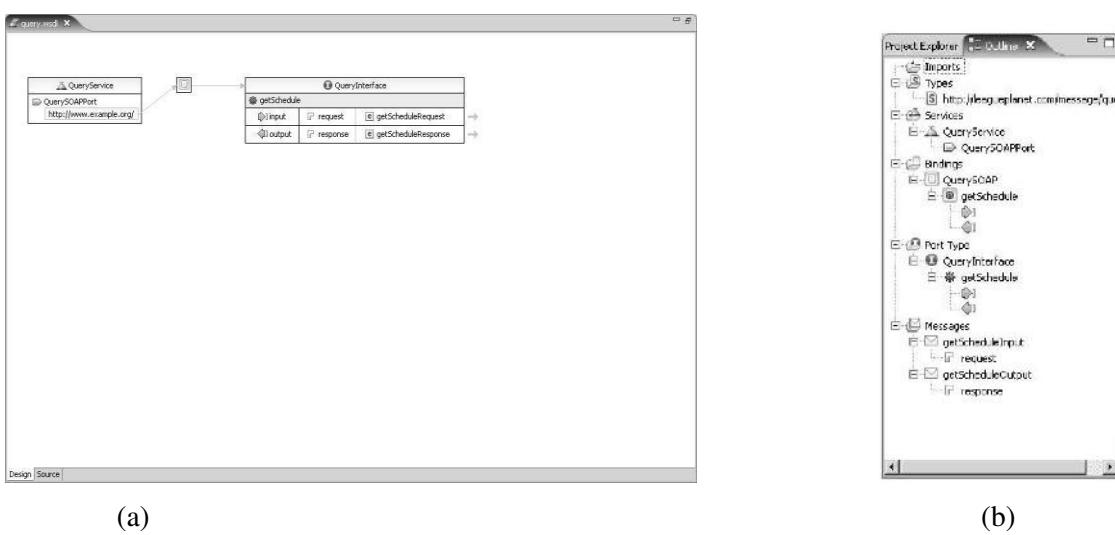
1. Create a new WSDL file with the extension .wsdl in the project (see Figure 3.60(a)).
2. Enter the namespace for the WSDL and have the wizard generate a skeleton document for you using the SOAP binding and document/literal style (see Figure 3.60(b)).



*Fig. 3.60. Screenshots of Eclipse interface - 4*

You can edit the document in the graph tab, the source tab, the outline view, and the property view. WSDL describes Web services using a hierarchy of constructs: message, portType, binding, and service. The editor has a wizard that generates binding content for you. Try to develop the wsdl file yourself.

View the wsdl file in the Design tab of the WSDL editor (see Figure 3.61(a)). View the wsdl file in the Outline view of the WSDL editor (see Figure 3.61(b)).

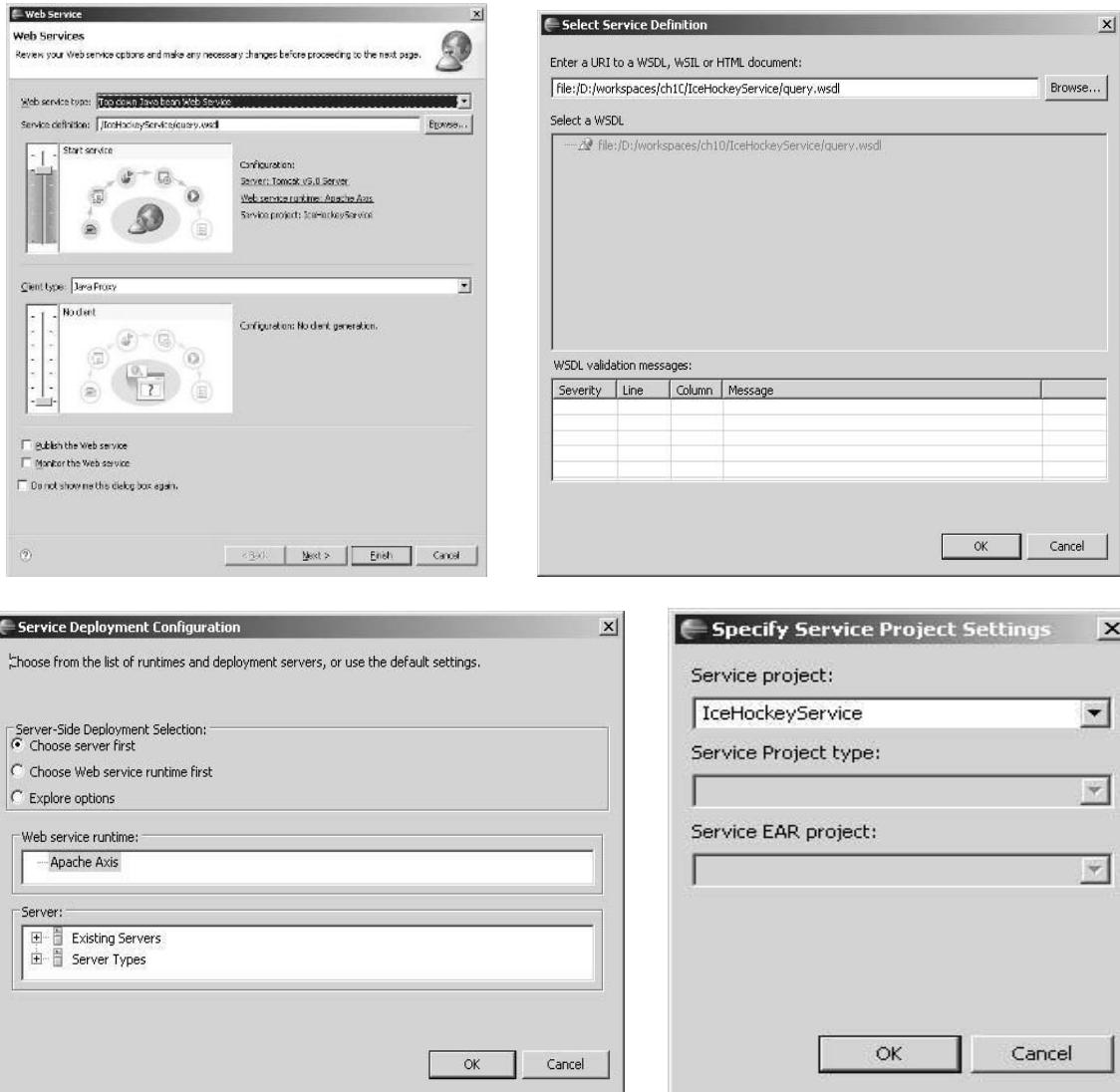


*Fig. 3.61. Screenshots of Eclipse interface - 5*

WTP provides a Web service wizard to simplify the task of **deploying Web** services. In the Top-Down approach, the WSDL document is used to define a Java server skeleton that implements the Web service. The Java server skeleton is deployed in a Web application. The wizard also sets up the Web application, copies a SOAP engine into it, Apache Axis for example, and generates any required deployment descriptors. All you need to do then is

fill in the implementation of the Java server skeleton with the business logic of the Web service.

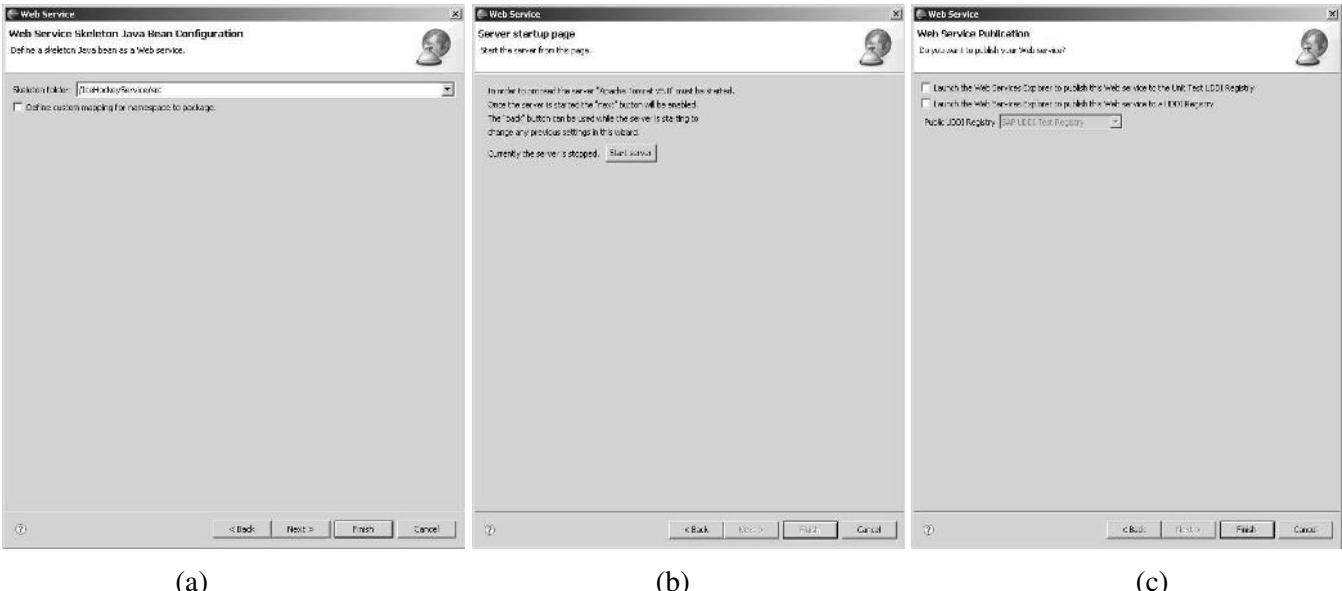
1. You have now described the Web service. Your next goal is to deploy it. This step assumes you have previously installed Tomcat and added it to WTP. Select the wsdl file and execute the command Web Services > Generate Java bean skeleton. This command launches the Web service wizard (see Figure 3.62(a)). Pull the Web service slider up to the Start service position.



**Fig.3.62. Screenshots of Eclipse interface - 6**

2. Since you selected wsdl file when you started the wizard, it appears as the Service definition. You can select a different WSDL file at this point by clicking the Browse button, which opens the Select Service Definition dialog (see Figure 3.62(b)). Click the OK button to keep the wsdl here.
3. The wizard selects Axis and Tomcat by default, which is what you'll use here. To change these, click the Server or Web service runtime links, which open the Service Deployment Configuration dialog (see Figure 3.62(c)). Click the OK button to dismiss the dialog and keep the current selections.
4. The wizard assumes that you are deploying the service to the same project as the WSDL file. To change this, click the Service project link, which opens the Specify Service Project Settings dialog (see Figure 3.62(d)). Click the OK button to keep the service.
5. Click the Next button to proceed. The wizard lets you select a sourcefolder and change the package name for the generated Java skeleton (see Figure 3.63(a)).
6. Click the Next button to proceed. The wizard is now ready to generate the code and deploy the Web service. The server must be started to complete this step (see Figure 3.63(b)).

7. Click the Start server button, wait until the server starts, and then click Next. The Web service is now deployed. Finally, the wizard lets you publish the WSDL to UDDI (see Figure 3.63(c)).



**Fig.3.63. Screenshots of Eclipse interface - 7**

There are a lot of steps involved in deploying a Web service. Fortunately, the Web service wizard handles all of these steps for you. In fact, if you are happy with the default behavior, you can simply click the Finish button on the first page of the wizard.

You can also avoid using the wizard altogether and use an Ant task that WTP provides instead. The Ant task is handy when you find yourself repeatedly using the wizard to redeploy a modified WSDL file.

The Web service wizard have done:

- installed the Axis SOAP engine in your dynamic Web project,
- generated the Java bean skeleton for your service, and lots of Java XML data binding classes in the src folder,
- copied query.wsdl to WebContent/wsdl/QuerySOAPPort.wsdl and set its endpoint address to your Web application (it also copied schedule.xsd),
- created the Axis deployment descriptor WebContent/WEB-INF/ server-config.wsdd,
- created a couple of handy Axis files to deploy and undeploy your Web service in a subfolder of WebContent/WEB-INF, and
- started Tomcat to make your Web service available.

To verify that the Web service is actually deployed and running, do the following:

1 Use the Project Explorer view to examine the project after the wizard completed (see Figure 3.64(a)). Note that the Axis runtime includes a servlet named AxisServlet, which lists the deployed Web services. Select the AxisServlet servlet and execute the Run As ->Run on Server command.

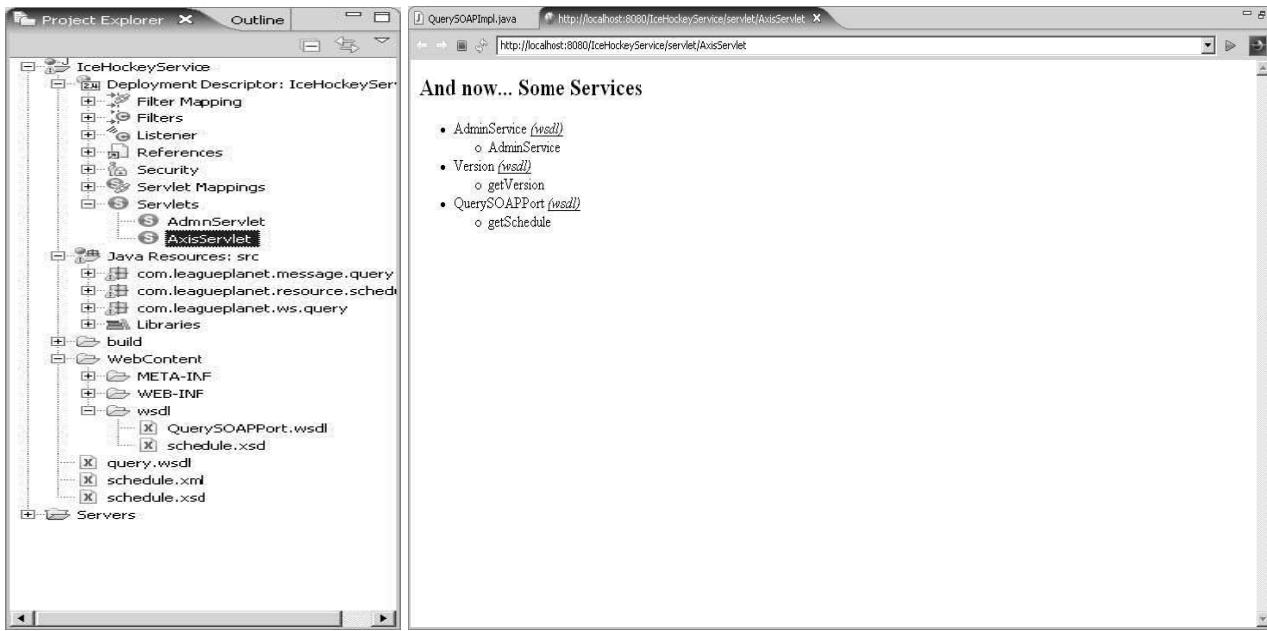
2 Running AxisServlet opens a Web browser with its URL. View the list of deployed Web services (see Figure 3.64(b)). Note that the new service appears in the list.

**Implementing the Web Service.** The Web service is running but it just returns null at this point. Next, you need to fill in the implementation of the Java bean skeleton. The Web service needs to access the service logic tier.

1 If you have not previously done so, create a new J2EE utility project and write the source code of the service logic in src. Now, make this project available to the Web service as follows: Select the project and open its Properties dialog. Add the new code as a J2EE Module Dependency (see Figure 3.65(a)).

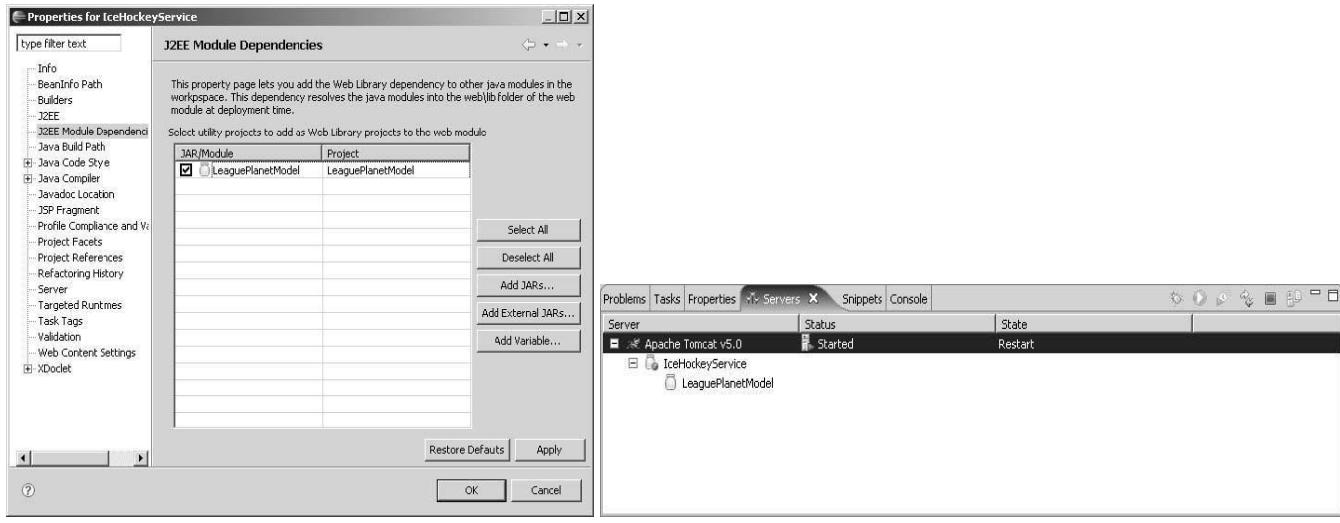
2 View the module structure of the server in the Servers view (see Figure 3.65(b)).

3 You now have a Web service skeleton and access to service logic tier. Your next goal is to implement the Web service. The name of the generated skeleton class ends with Impl. This should be modified to allow the access to the service logic tier.



(a)

(b)

*Fig. 3.64. Screenshots of Eclipse interface - 8*

(a)

(b)

*Fig. 3.65. Screenshots of Eclipse interface - 9*

The Web service uses classes that were generated from the xsd file. This might seem like a waste of effort, but it has a couple of big advantages. First, the classes generated from xsd serialize precisely into the XML format you defined, which means that, unlike the situation for SOAP encoding, other programs can process it interoperably using a wide range of XML processing techniques. This is the main advantage of the document/literal approach. Second, you are now free to change the business tier model without breaking clients of your Web service. They are completely decoupled from your internal implementation. This is the meaning of loose coupling. Of course, if you change the business tier model, then you'll also have to update the Web service implementation.

*Testing with the Web Services Explorer.* At this point the Web service is ready to test, and you will test it using the Web Services Explorer. The Web Services Explorer lets you test Web services without writing or generating any code. The Web Services Explorer accomplishes this by dynamically interpreting the WSDL for the Web service. You can test Web services that are deployed on your own machine or anywhere else on the Web.

The Web Services Explorer is itself a Web application. It runs in the embedded servlet container that Eclipse uses for displaying Help. The Web Services Explorer uses servlets and JSPs to generate its user interface, like any other Java Web application, but it is also integrated with Eclipse and can access the contents of your workspace.

Do the following to test your Web service:

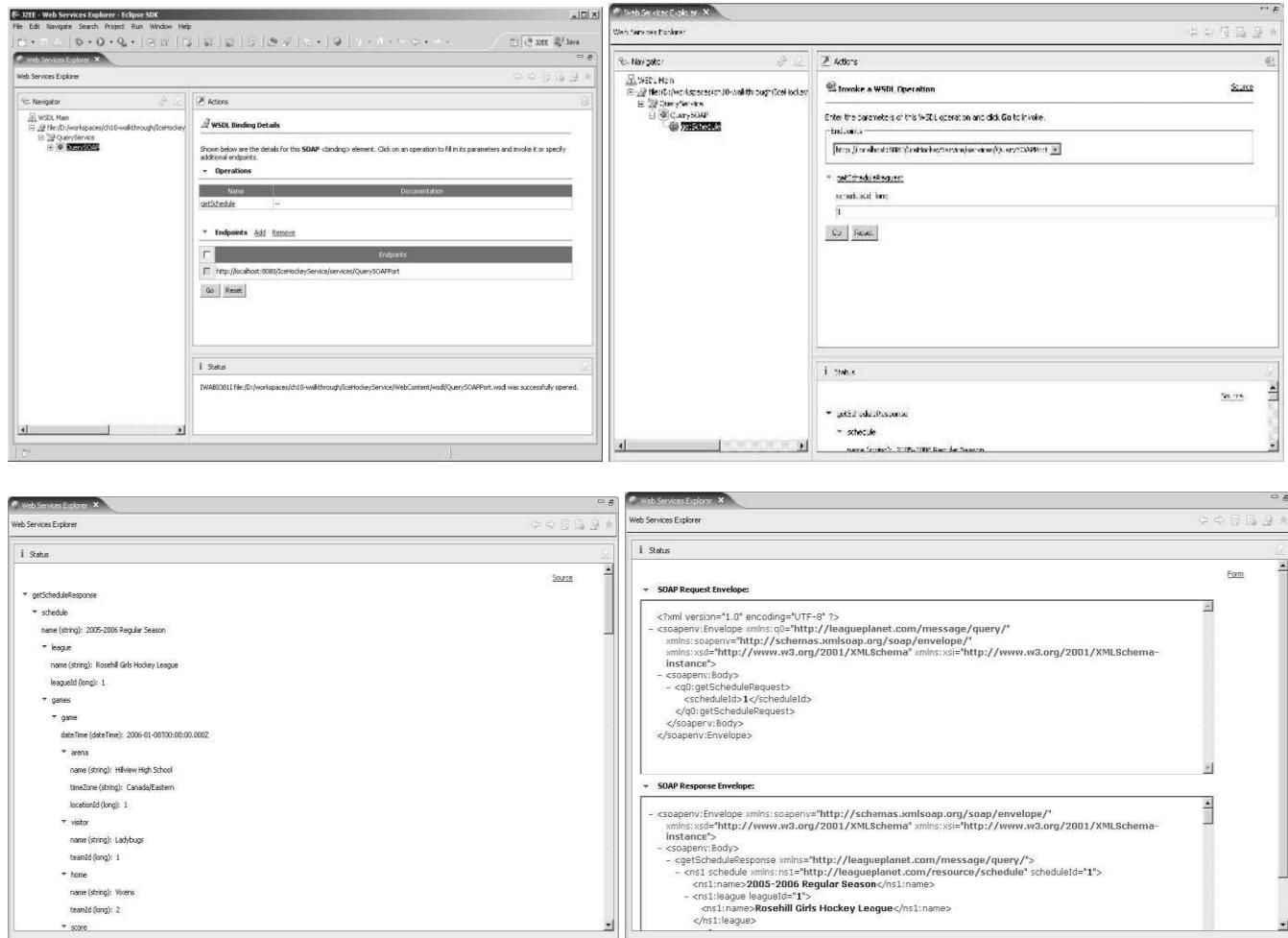
1. Select the service wsdl file and execute the command Web Services ->Test with Web Services Explorer. The Web Services Explorer will start and open a new Web browser in the editor area (see Figure 3.66(a)).

The Web Services Explorer user interface consists of three panes named Navigation, Action, and Status. The Navigation pane displays an object tree, which grows as you perform actions. The Action pane displays information about the currently selected object and lets you perform actions on it. The Status pane displays messages from the last performed action.

2 View a service operation (see Figure 3.66(b)). To test the operation, enter a value in the parameter field and click the Go button. The response is returned in the Status pane.

3 Double-click the Status pane title bar to maximize it (see Figure 3.66(c)). The Status pane displays the response from the Web service formatted as a form that hides the XML detail. The request and response can also be displayed in raw source format. Click the Source link to view the messages as raw XML SOAP envelopes.

4 View the SOAP message source (see Figure 3.66(d)). Click the Form link to return to the form display.



*Fig. 3.66 Screenshots of Eclipse interface - 10*

### 3.3.17.2 Iteration 2: Developing Web Services Bottom-Up

The Bottom-Up approach to Web service development begins with creation of a Java service class. The methods of the class define the operations of the Web service.

The argument lists and return types define the messages of the operations. After the service class is created, a tool is used to deploy it as a Web service and to generate the WSDL document that describes it. If changes are made to the interface of the class, the deployment and WSDL generation steps must be repeated.

The Bottom-Up approach lets Java developers become immediately productive at Web service development. No

new XSD and WSDL design skills are required. Bottom-Up development results in good Web service interfaces when the Java service class uses simple data transfer objects as the inputs and outputs of its operations. However, if complex objects are used, then the resulting XSD may be hard to understand and less interoperable. There is also the risk of “bleed-through” from the implementation into the service interface, which results in undesirable coupling between the client and service. If the Web service interface changes whenever you change the implementation of the Java service class, then you will continually break your clients and largely defeat the benefits of Web services.

The best way to create a clean, stable, interoperable Web service interface is to design the XSD for the messages first, and use the Top-Down approach. The next best way is to design a simple data transfer object layer for use in the Java service class interface, and use the Bottom-Up approach. If you do use the Bottom-Up approach, be disciplined about not changing the method signatures of the Java service class. Confine your changes to the method implementations to avoid breaking your clients.

In this iteration, you'll do the following:

- 1 Develop a Java service class to get details about a game and to update its score.
- 2 Use the Web service wizard to deploy the service.
- 3 Use the WSDL editor to view the generated WSDL.

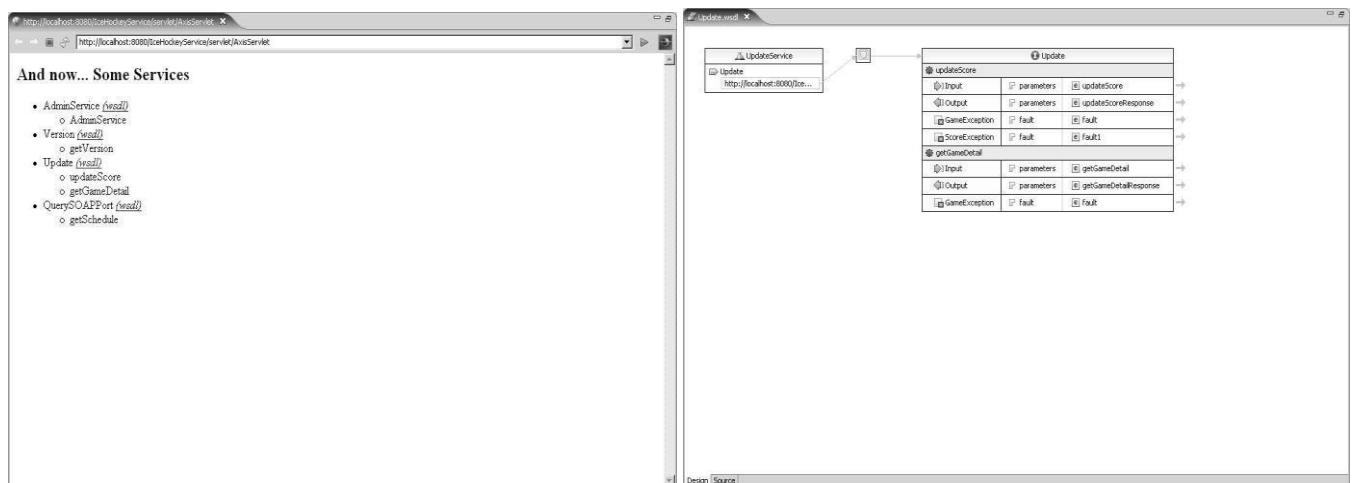
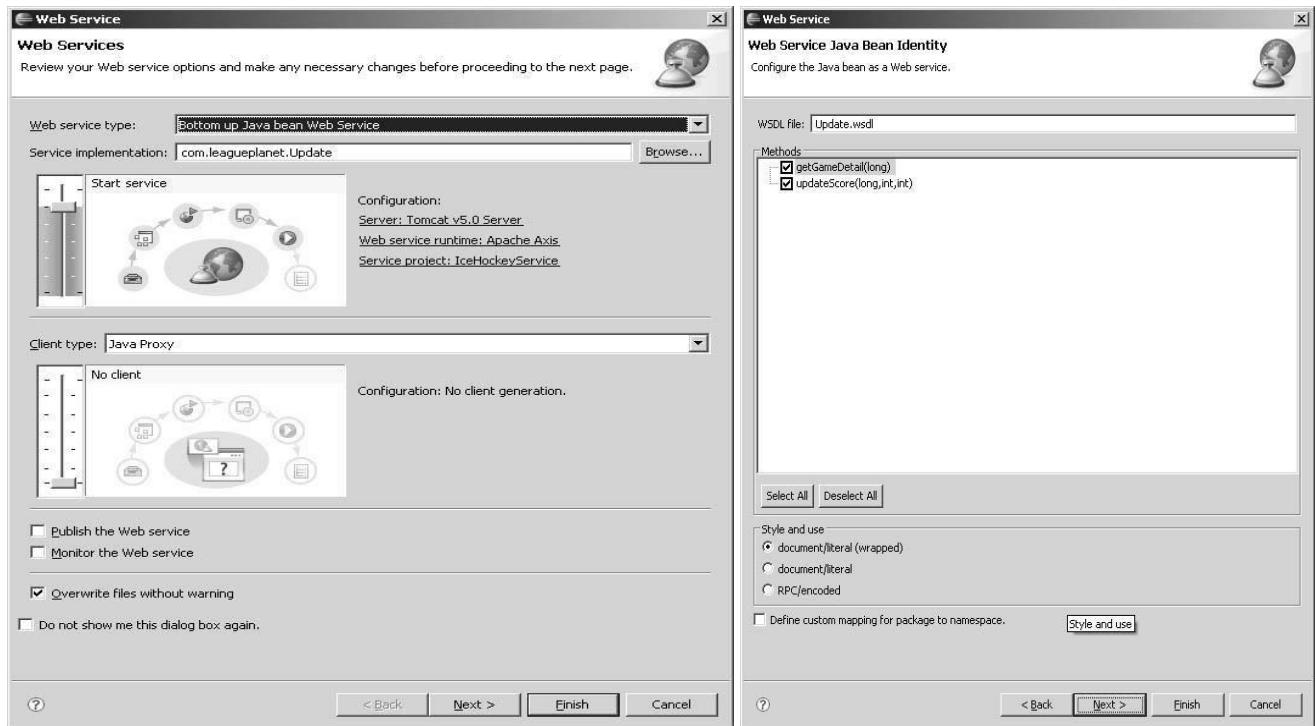
**Develop the Java Service Implementation.** Create a class and try your hand at designing it. A well-designed Web service should carefully validate its inputs and throw informative exceptions if the inputs are invalid. Errors in inputs should be detected at the earliest possible opportunity to simplify the task of problem diagnosis.

**Deploy the Service.** Deploying a Java class as a Web service is the process of adding it to the SOAP engine's configuration. Some aspects of this process are standardized and others are implementation dependent. Fortunately, the Web service wizard makes deploying Java classes easy. You simply select the Java service class and run the wizard. The wizard lets you control many aspects of how the service is deployed, tested, and published. To deploy the Update class, do the following:

1. Select the new class and execute the Web Services ->Create Web service command. The Web service wizard opens (see Figure 3.67(a)). Note that the Web service type is Bottom up Java bean Web Service since the Update class was selected when you invoked the wizard. Ensure that the service slider is at the Start service position and that the client slider is at the No client position. You could click the Finish button at this point since the wizard picks sensible defaults. Instead, click the Next button to step through the wizard pages.
2. The Java Bean Identity page appears (see Figure 3.67(b)). The wizard lets you select the methods to include in the Web service interface. The wizard also lets you specify a name for the generated WSDL file and control several aspects of how the WSDL is generated. One of the most important aspects of how the Web service is deployed is the *style/use* combination selected for the SOAP binding. The SOAP binding has *document* and *RPC* styles, and *literal* and *encoded* uses. *Document style* means that the SOAP body contains XML documents as children. *RPC style* means that the SOAP body conforms to a pattern that is used for remote procedure calls. *Literal use* means that the message content is literally described by the XML schema referenced by the WSDL document. *Encoded use* means that the message content conforms to the SOAP encoding specification and is only abstractly described by the XML schema referenced by the WSDL document.

Early SOAP implementations used the RPC/encoded combination, but this led to interoperability problems due to ambiguities in the SOAP encoding specification. The document/literal combination, with the additional WS-I recommendation that the SOAP body contain a single document child, is the preferred choice.

Although not formally specified anywhere, Microsoft introduced a pattern called *document/literal wrapped*, which can be used to generate a WS-I compliant document/literal WSDL document when deploying a service class in the Bottom-Up approach. In this pattern, the input message for an operation is composed of a document whose root element is the method name and whose child elements are the input parameters of the method. The output message is constructed similarly, except the root element is the con-catenation of the method name and a Response string.



**Fig. 3.67. Screenshots of Eclipse interface - 11**

Use document/literal wrapped for the best possible interoperability and ease of consumption by clients. Toolkits that recognize this pattern can generate client proxies whose interfaces match the service interface.

The wizard also lets you explicitly specify the namespace of the generated WSDL document. Accept the defaults and click the Finish button. The wizard deploys the Web service and generates the WSDL. You're now ready to verify that the service has been properly deployed.

3 The AxisServlet servlet gives you a handy way to verify that the Web service wizard succeeded in deploying your class. Select the AxisServlet servlet in the Project Navigator and execute the Run as ->Run on Server command. The Web browser opens on the AxisServlet servlet (see Figure 3.67(c)). Note that, as expected, the Web service is indeed now listed.

4 The wizard generated the wsdl document for the service. Open it in the WSDL editor and explore it (see Figure 3.67(d)). Note that the interface for the Web service contains the operations that match the methods of the Java service class.

### 3.3.17.3 Iteration 3: Generating Web Service Client Proxies

Web services can be invoked from programs written in many programming languages, including Java, C#, PHP, and JavaScript. Most languages have toolkits that support *dynamic invocation* of Web services and therefore do

not require any code generation. Dynamic invocation is very useful for cases where the WSDL of the Web service is not known in advance. For example, the Web Services Explorer is a general-purpose tool that can dynamically invoke any Web service given its WSDL at runtime.

However, for most application development purposes, the interface of the Web service is known at development time, although the endpoint at which the service is deployed may not be known until runtime. Web service toolkits typically include a code generation program, for example, Axis WSDL2Java, that can generate a client proxy from a WSDL document. A client proxy simplifies Web service invocation by providing a class that resembles the service interface. In J2EE, client proxies are specified by the JAX-RPC specification as well as its follow-on JAX-WS, which defines the binding between WSDL and Java.

The Web service wizard lets you generate a client proxy from a WSDL document. The wizard also includes the ability to generate test clients so you can immediately test the proxy. You can inspect the generated test client source code and copy useful snippets of it into your own application. In this iteration, you'll do the following:

- 1 Use the Web service wizard to generate a Java client proxy and a JSP test client for the Update service.
- 2 Test the Update service using the JSP test client.

**Generate a Java Client Proxy and JSP Test Client.** The Web service wizard helps you access and test Web services. To access a Web service, you select its WSDL document and generate a client proxy for it. The wizard is extensible so that code generators for any language can be added. Here you'll use the WSDL2Java code generator that is part of Apache Axis. The wizard also has an extension point for test facilities. You've already seen the use of the Web Services Explorer for testing Web services. Here you'll use a code generator that creates a JSP test client that invokes the generated Java client proxy.

1 Select the service wsdl file and execute the Web Services ->Generate Client command. The Web Service wizard opens (see Figure 3.68(a)). Pull the slider up to the Test client position and check the Monitor box.

2 You must generate the client proxy to a different project than the service to avoid filename conflicts. Click on the Client project link and set the output to be the desired name of the client project.

Always generate the client proxy into a different project than the deployed service to avoid filename conflicts. If you generate the client to the same project as the service and the Overwrite files without warning checkbox is checked, then the wizard will silently overwrite the service, causing it to fail.

3 Click the Next button. The Web Service Proxy Page appears (see Figure 3.68(b)). The wizard lets you change the source folder and package name for the generated client proxy code.

4 Click the Next button. The Server startup page appears (see Figure 3.68(c)). Click the Start server button and wait until the server starts.

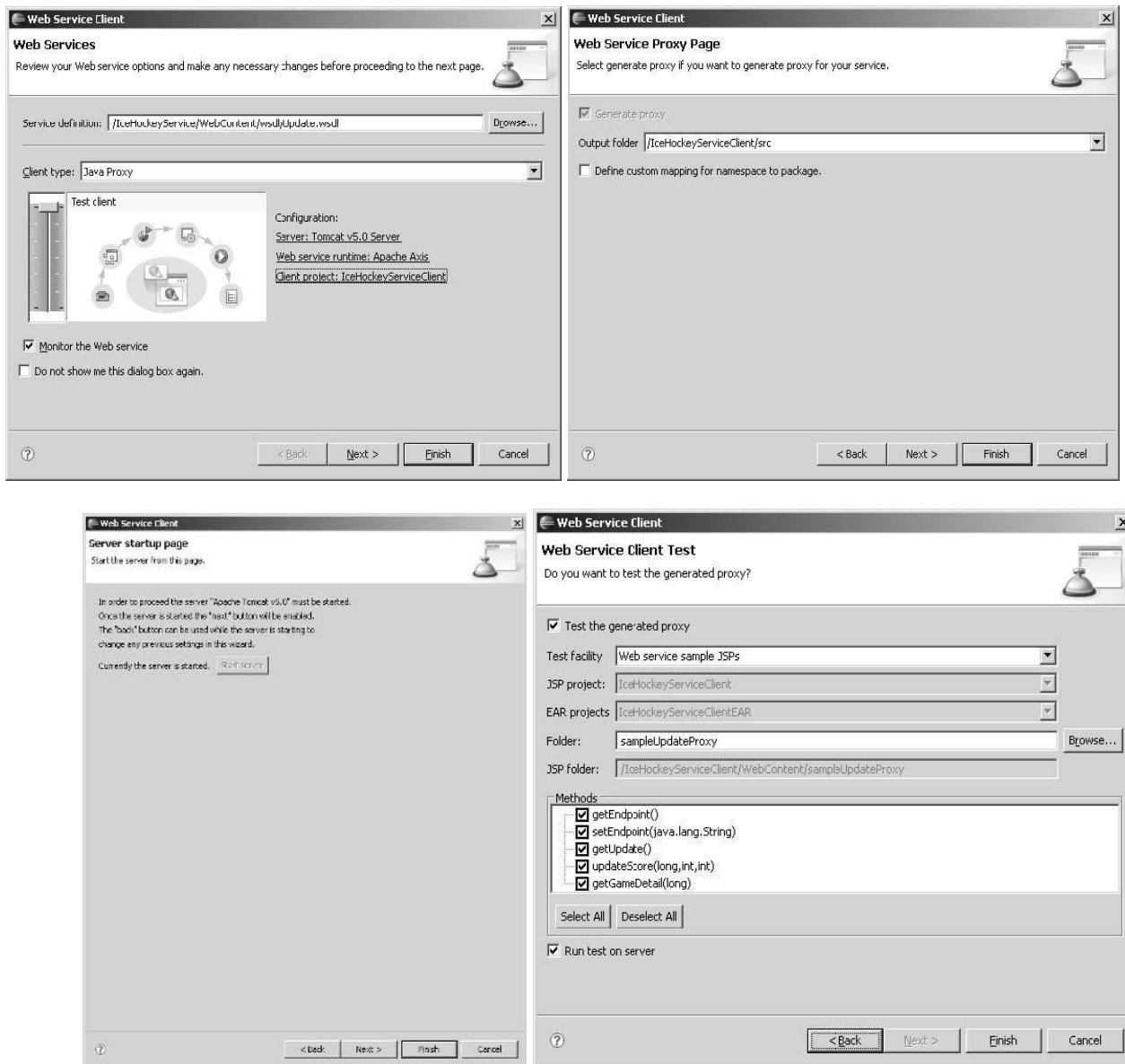
5 Click the Next button after the server has started. The Web Service Client Test page appears (see Figure 3.68(d)). The wizard lets you select the operations to include in the generated JSP test client. Note that the exposed methods have the same names as the Web service operations. These methods let your application invoke the corresponding Web service operations.

The exposed methods let you modify the Web service endpoint at runtime. These methods come in handy if you have to change the port number so you can send messages through the TCP/IP monitor.

The wizard also lets you select a different output folder for the JSPs. Click the Finish button. The wizard now generates code and launches a Web browser on the test client.

The Web service client wizard did a lot of work for you. It:

- created a new dynamic Web project,
- installed the Axis runtime libraries,
- generated Java proxy code, including XML data binding classes and exceptions, in the src folder,
- generated JSP test client code in the proxy subfolder of the WebContent folder of the client project,
- started an instance of the TCP/IP Monitor and configured the JSP test client endpoint to use it, and
- opened the JSP test client in a Web browser.

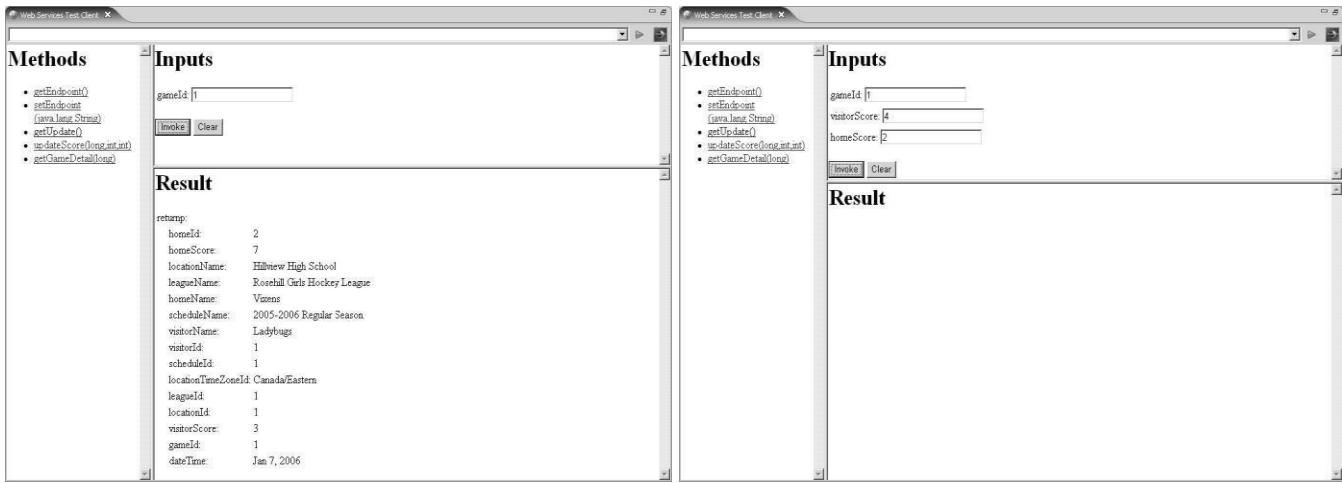


**Fig. 3.68. Screenshots of Eclipse interface - 12**

**Using the JSP Test Client.** The JSP test client is a Web application that lets you test a Web service using a Java client proxy. The user interface of the JSP test client has a Methods pane, an Inputs pane, and a Result pane. The Methods pane lists the methods of the Java client proxy, which include the operations of the Web service and the convenience methods for accessing the endpoint and the service class. When you click a method in the Methods pane, the Inputs pane is updated to display a data entry form for the input parameters of the selected method. The Inputs pane contains Invoke and Clear buttons. The Clear button clears the data entry form. The Invoke button invokes the selected method using the input parameter values that are entered in the Inputs pane. The result of the invocation is displayed in the Result pane. The Web service wizard opens the JSP test client in a Web browser. Start the test by getting the method parameters. The JSP test client invokes the Web service and receives the response. View the returned results the Result pane (see Figure 3.69(a-b)).

### 3.3.17.4 Iteration 4: Testing Web Services for Interoperability

WSs are designed to enable heterogeneous systems to interoperate over the Web. For example, you may deploy a service on a J2EE application server and want both .NET desktop clients and PHP Web clients to be able to access it. Previous distributed computing technologies differed from Web services either because they were designed for homogeneous systems or they used proprietary protocols. For example, Java RMI was designed to enable Java systems to interoperate, while Microsoft DCOM was designed for Windows to Windows communication. But the reality of the Web is that there is no single dominant technology. The Web is composed of a highly heterogeneous combination of hardware, operating systems, and programming languages.



**Fig. 3.69. Screenshots of Eclipse interface - 13**

Web services achieve interoperability by using XML, which is an architecturally neutral text format. However, this interoperability comes at a price since textual formats are less efficient than binary alternatives. Therefore if Web services fail to interoperate in practice, then we have paid the performance penalty for nothing.

The interoperability of the first wave of Web services was, in fact, disappointing, largely due to ambiguities, errors, and omissions in the initial SOAP 1.1 and WSDL 1.1 specifications. These specifications did not go through the rigorous standards development processes established by the W3C. The follow-on specifications, SOAP 1.2 and WSDL 2.0, corrected these deficiencies. However, the industry could not wait for these revisions and instead created the Web Services Interoperability Organization (WS-I) to fix the problem. WS-I issued the Basic Profile (BP) 1.0 to establish interoperability guidelines. One of the key recommendations of BP 1.0 was to use the document/literal binding for SOAP. BP 1.0 was later split into two specifications, the Simple SOAP Binding Profile (SSBP) 1.0 and the Attachments Profile (AP) 1.0.

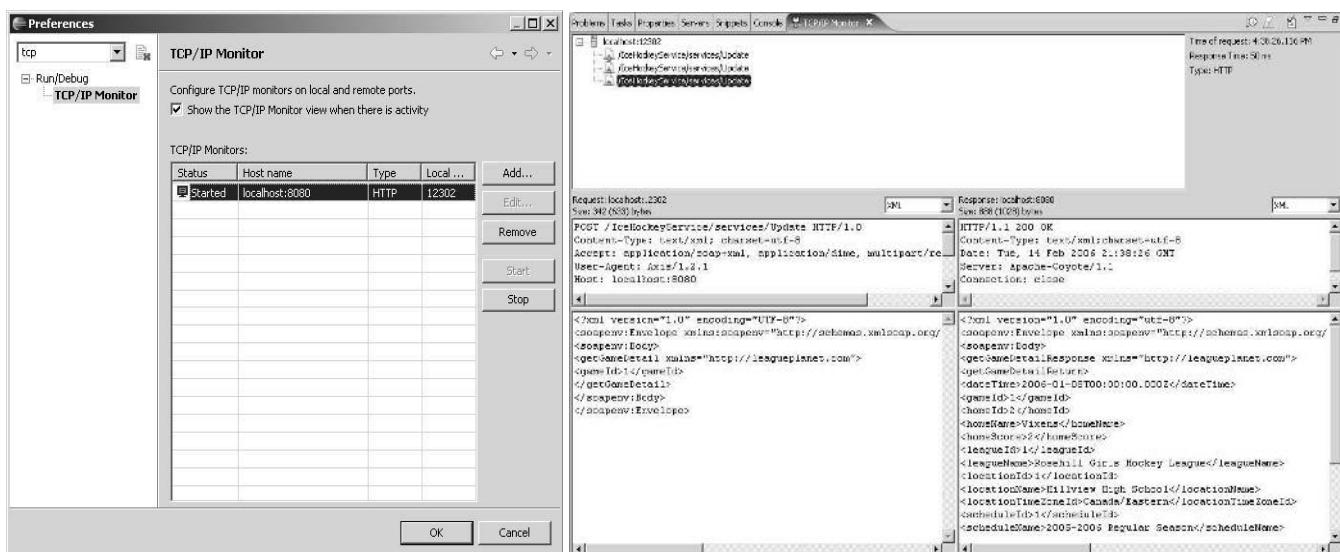
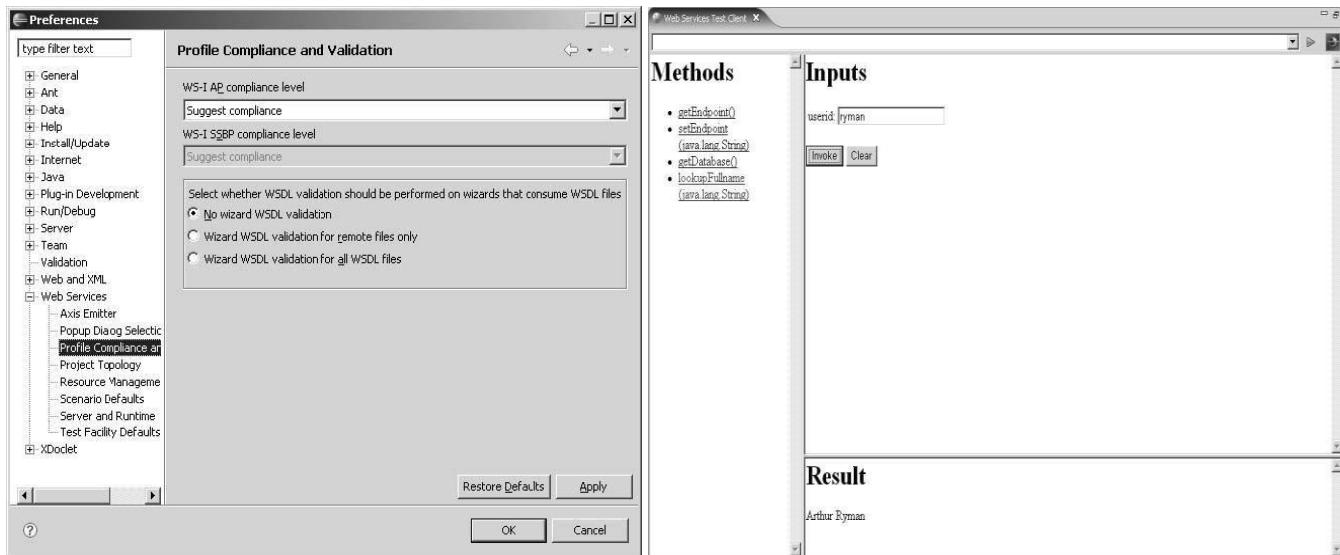
WTP includes WS-I Test Tools, which can validate HTTP SOAP messages and WSDL documents for WS-I compliance. These tools began life at WS-I as the reference Java implementation.

**Checking Messages for WS-I Compliance.** The WS-I Test Tools include two main components. The first component is an extension to the WSDL validator. You can check a WSDL document for WS-I compliance by enabling the WS-I compliance preferences and then validating the document as usual. The second component is a message log validator. This component is integrated with the TCP/IP monitor. You can save the messages captured by the monitor into an XML log file and run the message log validator on it to check for WS-I compliance. You'll be validating messages for WS-I compliance in this iteration. The WSDL and message log validation and WS-I compliance levels are specified in the Profile Compliance and Validation preference page (see Figure 3.70(a)).

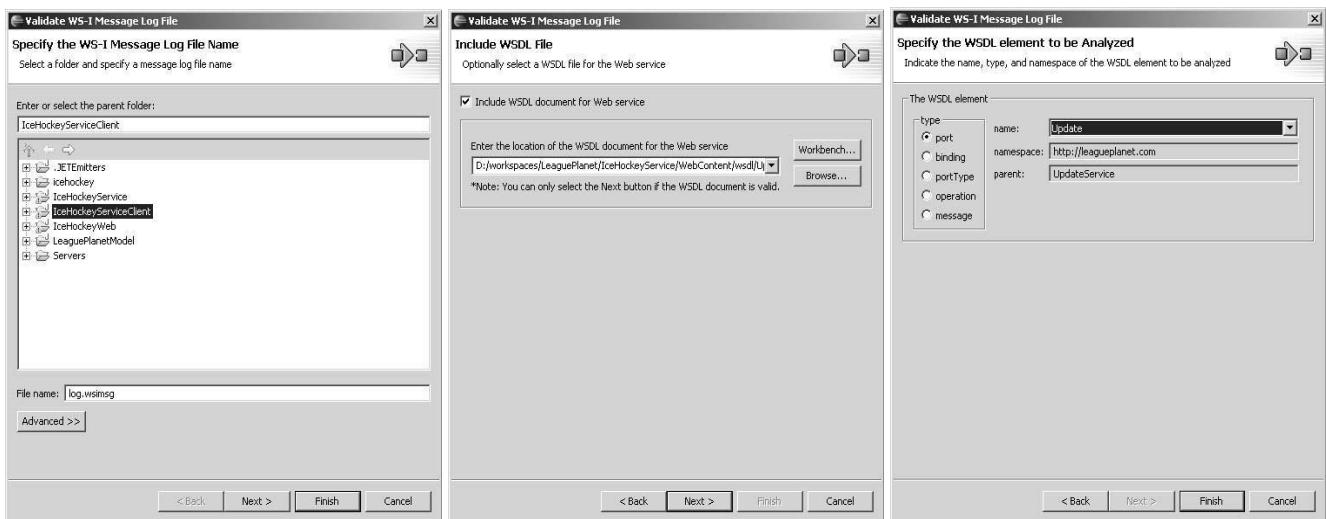
1. Return to the Web browser with the JSP test client running in it, which you launched in the previous iteration. If you already closed the JSP test client, select the `jsp` file in the proxy subfolder of the `WebContent` folder in the client project, and execute the `Run as Run on Server` command to open it. Click the `getEndpoint` method in the Methods pane. Click the `Invoke` button in the Inputs pane. View the endpoint address in the Result pane. Note the port number on the endpoint URL, for example, 12302, instead of the usual 8080 for Tomcat (see Figure 3.70(b)). This unusual port number is used by an instance of the TCP/IP monitor.

2. Open the Preferences dialog and select the TCP/IP Monitor page (see Figure 3.70(c)). Note that the port number of the monitor matches the port number of the endpoint. You can use this Preference page to manage TCP/IP monitor instances. A TCP/IP monitor instance listens to some port on localhost and forwards the requests to another, possibly remote, host and port. You can manually configure the JSP test client endpoint address to use a TCP/IP monitor instance by getting the current endpoint using the `getEndpoint` method and setting it to match the TCP/IP monitor port using the `setEndpoint` method.

3. View the recorded messages from Iteration 3 in the TCP/IP Monitor view (see Figure 3.70(d)). To validate the messages, click the Validate WS-I Message Log File icon (a document with checkmark) in the top right corner of the TCP/IP Monitor view.



**Fig. 3.70. Screenshots of Eclipse interface - 14**



**Fig. 3.71. Screenshots of Eclipse interface - 15**

4 The Validate WS-I Message Log File wizard opens (see Figure 3.71(a)). The messages are written into an XML log file. The wizard lets you select a folder to store the message log file. Select the client and click the Next button.

5 The Include WSDL File page appears (see Figure 3.72(b)). The wizard lets you optionally validate the message against a WSDL file. The messages should conform to the description in the wsdl document. Select wsdl file and click the Next button.

6 The WSDL Element page appears (see Figure 3.72(c)). The wizard lets you select the WSDL element to use. The message should conform to the description of the Update port element. Select the Update port and click the Finish button.

7 The wizard invokes the WS-I message log file validator and displays a success message since you selected WS-I compliant options when you deployed the Update class (see Figure 3.72(d)). If the validator found errors, it would place markers in the generated log file and these would appear in the Problems view as usual.

### 3.3.17.5 Iteration 5: Using Web Services in Web Applications

Web services can be used in applications developed in many popular programming languages and technologies. You should adhere to the WS-I guidelines to ensure that your Web services are consumable by the widest possible range of clients. You should also design your Web services to use XML messages that can be processed by a variety of programming technologies such as JAXB, DOM, SAX, StAX, and XSLT.

Web services allow alternate user interfaces and applications to be developed. A Web service interface lets other parties develop alternate, say, .NET desktop clients.

Web services also allow decoupling of the presentation and business tiers within an enterprise. For example, you could host the presentation and business tiers on different physical servers and drive the presentation tier off a Web service interface on the business tier. This decoupling allows the two tiers to be developed by different teams, at different times, using different programming technologies. For example, the business tier could be developed using J2EE, and the presentation tier could be developed using PHP or AJAX. The WSDL documents that describe the Web service interface on the business tier act like a contract between the tiers and insulate them from changes in implementation technology.

Java applications can use JAX-RPC or JAX-WS to access both Java and non-Java Web services. In this iteration, you'll develop a Java Web application that accesses the Web service interface of its business tier.

In this iteration, you'll:

- 1 Generate a Java client proxy for the Web service.
- 2 Develop a user interface based on JSPs and servlets.
- 3 Access the Web services from the servlets using the JAX-RPC programming model.
- 4 Run the Web application.

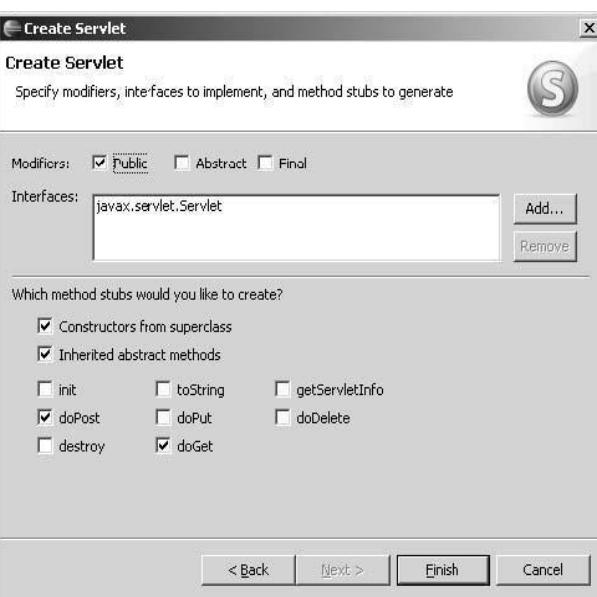
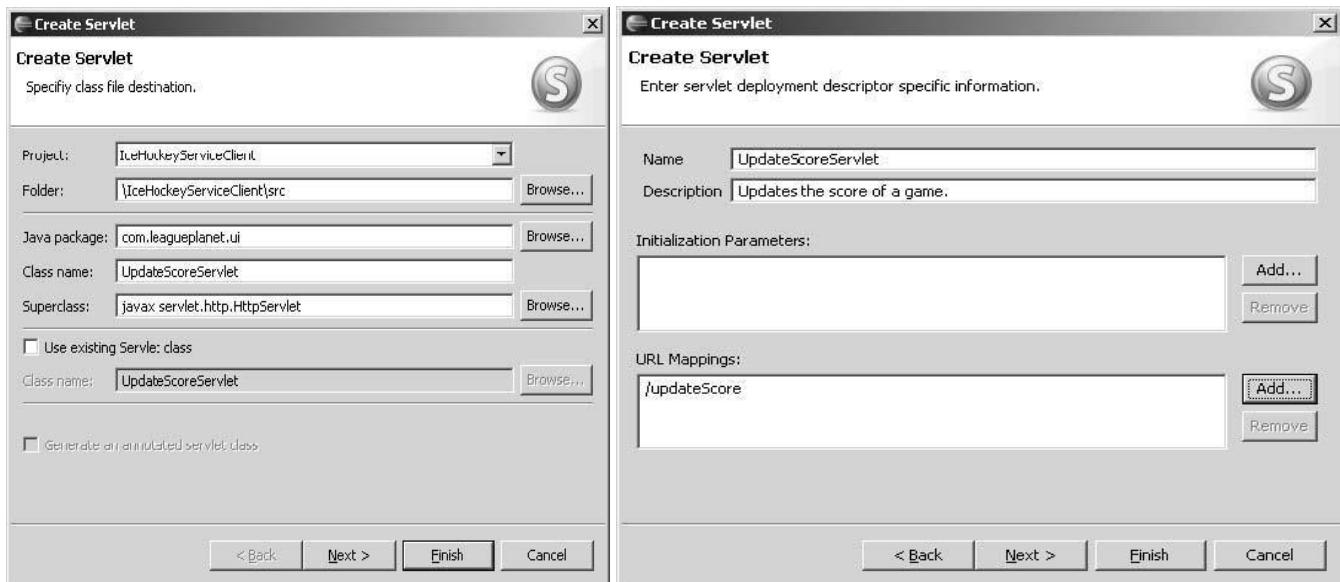
**Generate the Query Web Service Client.** Repeat the steps in Iteration 3 to generate a Java client proxy and JSP test client. Run the JSP test client to verify that it is working correctly. You should now have two Java client proxies—one for updating scores and another for getting schedules—in the client project, which is where you will build the user interface for your Web application.

**Create the Servlets.** Create these using the New Servlet wizard as follows:

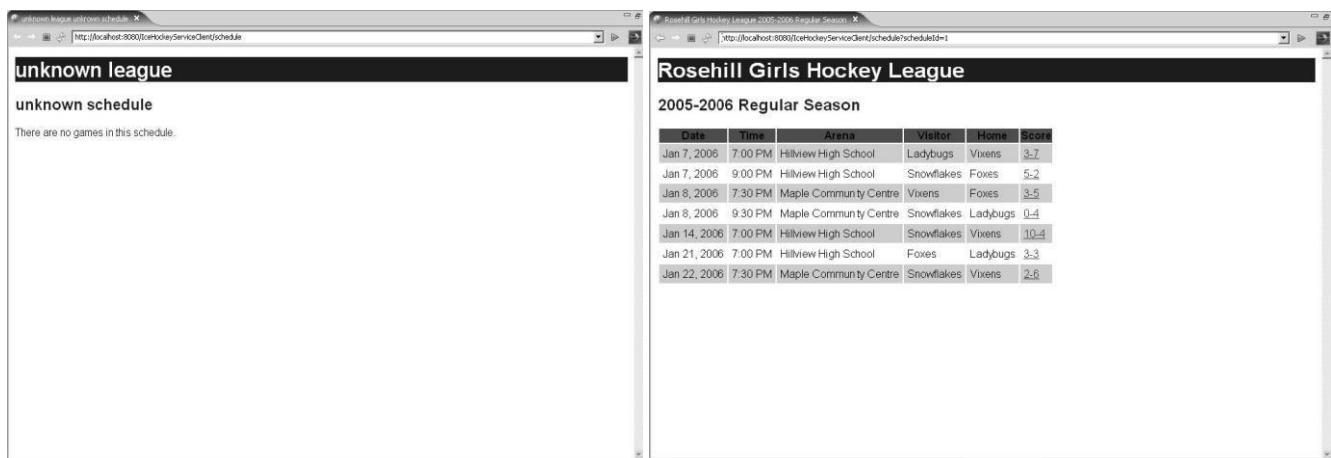
1 Select the client project and invoke the New ->Servlet wizard. The class file destination page appears (see Figure 3.72(a)).

2 The deployment descriptor information page appears (see Figure 3.72(b)). In the Description field enter a text about the servlet. In the URL Mappings field enter the name of a specific directory. In general, it's good planning to use URLs that don't reveal the implementation technology in case you want to change it later.

3 The class structure page appears (see Figure 3.72(c)). Check the methods boxes. The wizard creates the servlet, updates the deployment descriptor, and opens a Java editor on the servlet class.



*Fig. 3.72. Screenshots of Eclipse interface - 16*



*Fig. 3.73. Screenshots of Eclipse interface - 17*

Note the JAX-RPC client programming model in the exposed methods. The locator class is instantiated to get an object that implements the service interface. This object represents an instance of the service.

### **Test the User Interface**

1 Select the schedule servlet and execute the Run As ->Run on Server command. The Web browser displays a page titled the ext that was introduced (see Figure 3.73(a)).

2 Append the query string ?name\_of\_the\_parameter=value to the URL in the Web browser and reload the page. This time the schedule servlet handles the GET request by calling the operation of the Web service, adding the returned object to the session, and then forwarding the request to the jsp to display the result (see Figure 3.73(b)).

### **3.3.17.6 Iteration 6: Discovering and Publishing Web Services**

The Web started small. It was invented at CERN, a high-energy particle physics lab, as a way for scientists to share information. At first there was only a handful of Web sites, so finding what you were looking for was not a problem. Then the rest of the world discovered the Web and the number of sites exploded. The difficulty of finding information on the Web gave birth to indexers like Yahoo. Site owners entered descriptions of their content in a hierarchical classification scheme so Web surfers could do searches. This approach worked for a while, but as the number of sites grew and the rate of change of content accelerated, an automated approach was needed. Web crawlers such as Lycos were created to automatically transverse the Web and index the content of pages. Site owners could assist the Web crawlers by publishing metadata in robots.txt files, which listed the root pages to crawl. Today Google represents the pinnacle of Web-crawling technology. It's hard to imagine what the Web would be like without it.

This story is being replayed for Web services, although on a much smaller scale. Web service *discovery* is the task of locating Web services that perform a desired function and that satisfy other criteria such as quality of service or geo-graphic location. For example, you may want to locate a flower delivery service that is located in Gladstone, Australia, so that you can send your mother-in-law roses on her birthday. Web service *publication* is the task of making information about a Web service available so that it can be indexed or searched.

In this iteration you will use two technologies for the discovery and publication of Web services: Universal Description, Discovery, and Integration (UDDI); and Web Service Inspection Language (WSIL), which is also referred to as WS-Inspection. UDDI is a registry technology that has programmatic interfaces for publishing and querying information about Web services. UDDI is therefore analogous to the original Yahoo index. WSIL is a simple XML file format for listing Web services. WSIL uses root XML files, named inspection.wsil by convention, that are analogous to the robots.txt files that guide Web crawlers. UDDI and WSIL are complementary in that a Web service crawler could automatically populate a UDDI registry using information retrieved from inspection.wsil files.

In this iteration, you will do the following:

- 1 Search a UDDI registry for Web services.
- 2 Browse a WSIL document that lists Web services.
- 3 Create a WSIL document to describe the League Planet Web services.

The **UDDI** business registry standard was created in anticipation of the need to publish information about large numbers of Web services. With UDDI, Web service owners register and classify their services through a *publishing* interface. Developers or programmatic agents that are searching for Web services can then query UDDI registries through an *inquiry* interface. These interfaces are themselves made available as SOAP Web services. UDDI registries typically provide a Web user interface that lets users manually publish and query Web service information. Developers can programmatically access UDDI registries through toolkits such as UDDI4J. The JAXR specification is the Java standard for access to UDDI and other registries. WTP provides the Web Services Explorer, which is a Web application based on UDDI4J that acts as a universal client to UDDI registries. The Web Services Explorer lets you flow Web service information seamlessly between UDDI registries and your Eclipse workspace.

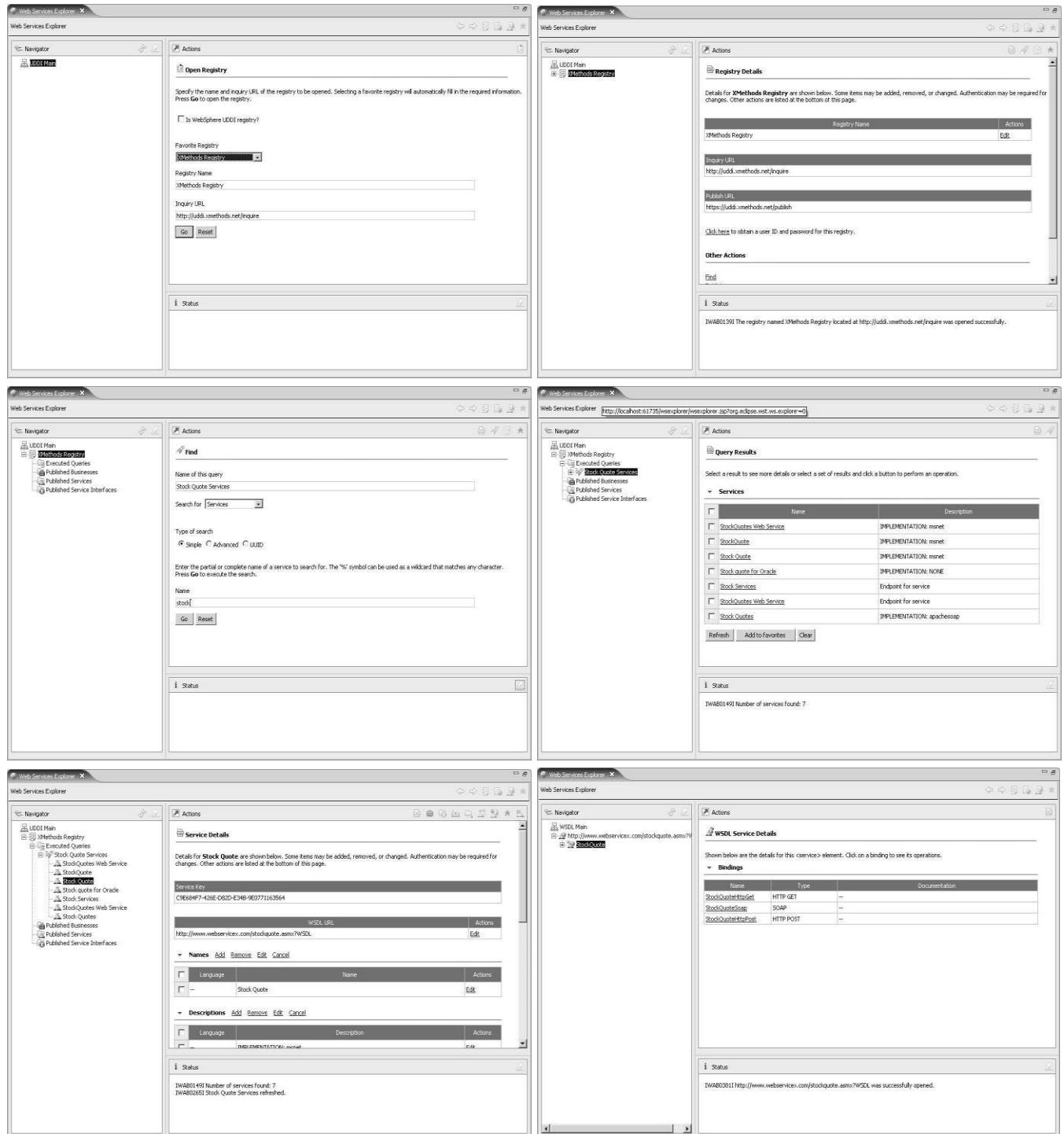
It was further proposed that there would be a network of public UDDI registries, sometimes referred to as the *UDDI cloud*, that were linked to each other and that replicated information between each other. A Web service registered in one registry would be replicated to all other registries in the network. Therefore, any of the linked registries could be queried to locate any service no matter where it was initially registered.

Although the network of public UDDI registries was built, it did not achieve much market acceptance and has since been dismantled. Perhaps the number of publicly available Web services did not grow to the point where a

registry was needed. Or perhaps the burden of registering services was too onerous. Maybe a scheme based on Web service crawlers would have succeeded. In any case, the use of UDDI now seems confined to within enterprises where it serves as a central place to register and locate in-house Web services.

Nevertheless, there is a very interesting publicly accessible UDDI registry at XMethods. This registry is not replicated with other registries, but it has become a place where many Web service developers advertise their work. To explore the XMethods UDDI registry, do the following:

- 1 Launch the Web Services Explorer by clicking its icon in the J2EE perspective or executing the Import ->Other ->Web Service command. The Web Services Explorer appears in a Web browser with the Open Registry page displayed (see Figure 3.74(a)). The Web Services Explorer user interface is divided into Navigator, Actions, and Status panes. The Navigator pane displays an object history tree for either UDDI, WSIL, or WSDL. The Actions pane displays a form for the currently selected object. The Status pane displays the results of the last performed action. In the Actions pane, select the XMethods UDDI Registry and click the Go button.



**Fig. 3.74 Screenshots of Eclipse interface - 18**

- 2 The Registry Details page is displayed (see Figure 3.74(b)). View the registry details. Click Find link.
- 3 The Find page is displayed (see Figure 3.74(c)). Enter Stock Quote Services as the name for the query. Search for Services. Enter a partial service name stock to search for and click the Go button.
- 4 The Query Results page is displayed (see Figure 3.74(d)). View the query results. Click the service links to explore the services. Click the link for the Stock Quote service and continue.
- 5 The Service Details page is displayed (see Figure 3.74(e)). View the webservicex.com service details. Click the Add to WSDL Page icon (the one with the plus sign) in the top right corner of the Actions pane to explore the WSDL document for this service.
- 6 The WSDL Service Details page is displayed (see Figure 3.74(f)). This page lists the bindings for the selected service. View the WSDL service details. Click the StockQuoteSOAP binding link.
- 7 The WSDL Binding Details page is displayed (see Figure 3.75(a)). This page lists the operations for the selected binding. View the WSDL binding details. Click the GetQuote operation link.
- 8 The Invoke a WSDL Operation page is displayed (see Figure 3.75(b)). This page lists the inputs for the selected operation. View the operation details. Click the Add link and enter a stock symbol. Click the Go button.
- 9 The Web Services Explorer invokes the operation and displays the result in the Status pane (see Figure 3.75(c)). View the result of the operation in the Status pane. Double-click on the title of the Status pane to maximize it. Click the Source link to view the request and response SOAP messages.



**Fig. 3.75. Screenshots of Eclipse interface - 19**

**WSIL** is a much simpler way, to publish information about Web services, than UDDI. WSIL is an XML format that you publish on your Web site to advertise available Web services. WSIL documents can refer to WSDL, UDDI, and other WSIL documents. By convention, the root WSIL document for a Web site is named

inspection.wsil. It can directly list all the Web services or point to subordinate WSIL documents. In the future, Web service crawlers might search the Web for inspection.wsil files and automatically index them in UDDI or other registries.

WSIL was jointly developed by IBM and Microsoft, but Microsoft still uses the precursor DISCO format to publish Web service information. There is not a lot of WSIL deployed at present. However, XMethods supports it and several other Web service publication technologies.

In this part of the iteration you will use the Web Services Explorer to view a WSIL document published at XMethods. You will also use WTP to create your own WSIL document to publish a Web service. Do the following:

- 1 Open a Web browser and surf to

<http://www.xmethods.net>

The XMethods home page is displayed (see Figure 3.76(a)). Look at the Programmatic Interfaces section, which lists UDDI, WS-Inspection, DISCO, RSS, and SOAP. These are the ways that XMethods publishes Web service information. Click the Access link.

**Screenshot (a): XMethods Home Page**

Welcome to XMethods. **Programmatic Interfaces**

Access XMethods through a variety of interfaces:

- UDDI v2
- WS-Inspection
- RSS
- SOAP
- DISCO

**Screenshot (b): Programmatic Interfaces to XMethods**

In addition to the browser interface, XMethods publishes a number of programmatic interfaces. The following table provides a catalog of these interfaces.

Interface	URL
XMethods SOAP Interfaces	<a href="http://www.xmethods.net/wsdl/query.wsdl">http://www.xmethods.net/wsdl/query.wsdl</a>
XMethods UDDI Private Registry	<a href="http://uddi.xmethods.net/inspire">http://uddi.xmethods.net/inspire</a>
WS-Inspection document	<a href="http://www.xmethods.net/inspection.wsil">http://www.xmethods.net/inspection.wsil</a>
DISCO document	<a href="http://www.xmethods.net/default.disco">http://www.xmethods.net/default.disco</a>
RSS feed	<a href="http://www.xmethods.net/interfaces/rss">http://www.xmethods.net/interfaces/rss</a>

**Screenshot (c): Web Services Explorer - WSIL Main**

Enter the URL of a WSIL document or a WSDL document and click Go to open inspect.

URI to document: <http://www.xmethods.net/inspection.wsil>

Choose the type of objects to inspect: **WSDL Services**

Go Reset

**Fig. 3.76 Screenshots of Eclipse interface - 20**

- 2 The Programmatic Interfaces to XMethods page is displayed (see Figure 3.76(b)). View the many access methods supported by XMethods. Copy the WS-Inspection link, which gives the URL to the inspection.wsil document.

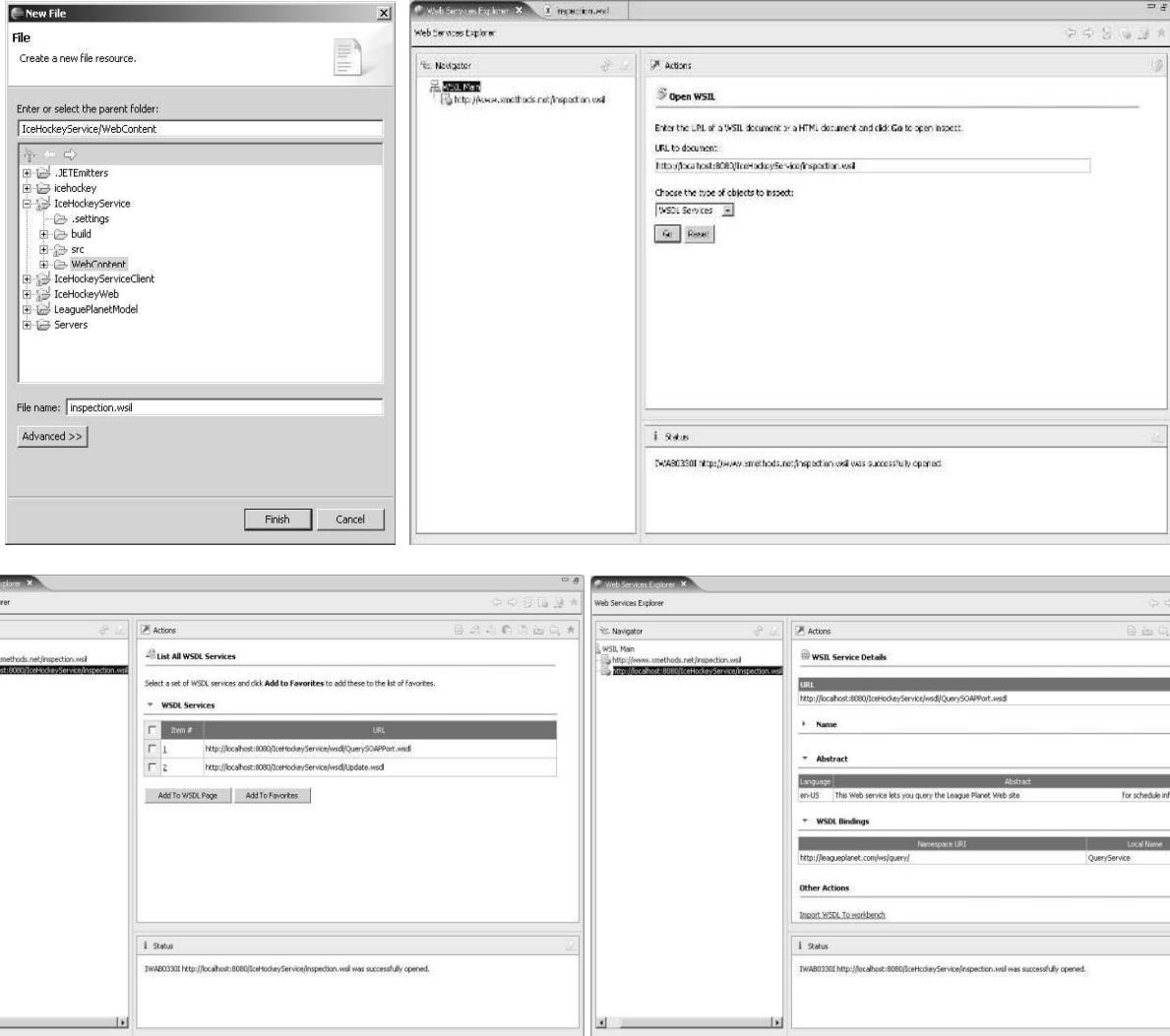
- 3 Click the WSIL Page icon (the page with globe) at the top right corner of the Web Services Explorer to open the WSIL page in the Navigator pane (see Figure 3.76(c)). Paste in the XMethods WSIL URL

<http://www.xmethods.net/inspection.wsil>

select WSDL Services, and click the Go button.

4 The List All WSDL Services page is displayed (see Figure 3.76(d)). View the list of WSDL services registered at XMMethods.

5 You are now going to create your own WSIL document for a specific Web service. Use the New File wizard to create a new inspection.wsil file in the service/WebContent folder (see Figure 3.77(a)).



**Fig. 3.77 Screenshots of Eclipse interface – 21**

6 You will now use WTP to generate WSIL file for the given Web service. Select the port-wsdl and execute the Web Services ->Generate WSIL command to create the port-wsil file. Save the file as inspection.wsil and add abstracts to describe them.

7 Enter the URL

`http://localhost:8080/service_name/inspection.wsil`

in the Web Services Explorer WSIL page, select WSDL Services and click Go (see Figure 3.77(b)).

8 The List All WSDL Services page is displayed (see Figure 3.77(c)). View the WSDL services available and click the port-wsdl link.

9 The WSIL Service Details page is displayed (see Figure 3.77(d)). View the port-wsdl details.

### **3.3.18 Web Service Support in .NET**

The .NET framework is a proprietary solution runtime and development platform designed for use with Windows operating systems and server products. The .NET platform can be used to deliver a variety of applications, ranging from desktop and mobile systems to distributed Web solutions and Web services.

A primary part of .NET relevant to SOA is the ASP.NET environment, used to deliver the Web Technology layer within SOA (and further supplemented by the Web Services Enhancements (WSE) extension).

**Architecture components.** The .NET framework provides an environment designed for the delivery of different types of distributed solutions. Listed here are the components most associated with Web-based .NET applications:

- ASP.NET Web Forms. These are dynamically built Web pages that reside on the Web server and support the creation of interactive online forms through the use of a series of server-side controls responsible for auto-generating Web page content.
- ASP.NET Web Services. An ASP.NET application designed as a service provider that also resides on the Web server.
- Assemblies. An assembly is the standard unit of processing logic within the .NET environment. An assembly can contain multiple classes that further partition code using object-oriented principles. The application logic behind a .NET Web service is typically contained within an assembly (but does not need to be).

ASP.NET Web Forms can be used to build the presentation layer of a service-oriented solution, but it is the latter two components that are of immediate relevance to building Web services.

**Runtime environment.** The architecture components previously described rely on the Common Language Runtime (CLR) provided by the .NET framework. CLR supplies a collection of runtime agents that provide a number of services for managing .NET applications, including cross-language support, central data typing, and object lifecycle and memory management.

Various supplementary runtime layers can be added to the CLR. ASP.NET itself provides a set of runtime services that establish the HTTP Pipeline, an environment comprised of system service agents that include HTTP modules and HTTP handlers. Also worth noting is that the established COM+ runtime provides a further set of services (including object pooling, transactions, queued components, and just-in-time activation) that are made available to .NET applications.

The .NET framework provides unified support for a set of **programming languages**, including Visual Basic, C++, and the more recent C#. The .NET versions of these languages have been designed in alignment with the CLR. This means that regardless of the .NET language used, programming code is converted into a standardized format known as the Microsoft Intermediate Language (MSIL). It is the MSIL code that eventually is executed within the CLR.

.NET provides programmatic access to numerous framework (operating system) level functions via the .NET Class Library, a large set of APIs organized into namespaces. Each namespace must be explicitly referenced for application programming logic to utilize its underlying features.

**APIs.** Following are examples of the primary namespaces that provide APIs relevant to Web services development:

- System.Xml. Parsing and processing functions related to XML documents are provided by this collection of classes. Examples include: The `XmlReader` and `XmlWriter` classes that provide functionality for retrieving and generating XML document content; Fine-grained classes that represent specific parts of XML documents, such as the `XmlNode`, `XmLElement`, and `XmlAttribute` classes.
- System.Web.Services. This library contains a family of classes that break down the various documents that comprise and support the Web service interface and interaction layer on the Web server into more granular classes. For example, WSDL documents are represented by a series of classes that fall under the `System.Web.Services.Description` namespace. Communication protocol-related functionality (including SOAP message documents) are expressed through a number of classes as part of the `System.Web.Services.Protocols` namespace. The parent `System.Web.Services` class that establishes the root namespace also represents a set of classes that express the primary parts of ASP.NET Web service objects (most notably, the `System.Web.Services.WebService` class). Also worth noting is the `SoapHeader` class provided by the `System.Web.Services.Protocols` namespace, which allows for the processing of standard SOAP header blocks.

In support of Web services and related XML document processing, a number of additional namespaces provide class families, including:

- `System.Xml.Xsl` Supplies documentation transformation functions via classes that expose XSLT-compliant features.
- `System.Xml.Schema` A set of classes that represent XML Schema Definition Language (XSD)-compliant features.
- `System.Web.Services.Discovery` Allows for the programmatic discovery of Web service metadata.

.NET **service providers** are Web services that exist as a special variation of ASP.NET applications, called ASP.NET Web Services. You can recognize a URL pointing to an ASP.NET Web Service by the ".asmx" extension used to identify the part of the service that acts as the endpoint. ASP.NET Web Services can exist solely of an ASMX file containing inline code and special directives, but they are more commonly comprised of an ASMX endpoint and a compiled assembly separately housing the business logic.

To support the creation of service requestors, .NET provides a proxy class that resides alongside the **service requestor's** application logic and duplicates the service provider interface. This allows the service requestor to interact with the proxy class locally, while delegating all remote processing and message marshalling activities to the proxy logic.

The .NET proxy translates method calls into HTTP requests and subsequently converts the response messages issued by the service provider back into native method return calls.

The code behind a proxy class is auto-generated using Visual Studio or the WSDL.exe command line utility. Either option derives the class interface from the service provider WSDL definition and then compiles the proxy class into a DLL.

The ASP.NET environment utilizes many system-level **agents** that perform various runtime processing tasks. As mentioned earlier, the ASP.NET runtime outfits the HTTP Pipeline with a series of HTTP Modules. These service agents are capable of performing system tasks such as authentication, authorization, and state management. Custom HTTP Modules also can be created to perform various processing tasks prior and subsequent to endpoint contact. Also worth noting are HTTP Handlers, which primarily are responsible for acting as runtime endpoints that provide request processing according to message type. As with HTTP Modules, HTTP Handlers can also be customized. Other parts of the HTTP Pipeline not discussed here include the HTTP Context, HTTP Runtime, and HTTP Application components.

Another example of service agents used to process SOAP headers are the filter agents provided by the WSE toolkit (officially called WSE filters). WSE provides a number of extensions that perform runtime processing on SOAP headers. WSE therefore can be implemented through input and output filters that are responsible for reading and writing SOAP headers in conjunction with ASP.NET Web proxies and Web services. WSE filters position themselves to intercept SOAP messages after submission on the service provider's end and prior to receipt on the service requestor's side.

**Platform extensions.** The Web Services Enhancements (WSE) is a toolkit that establishes an extension to the .NET framework providing a set of supplementary classes geared specifically to support key WS-\* specification features. It is designed for use with Visual Studio and currently promotes support for the following WS-\* specifications: WS-Addressing, WS-Policy, WS-Security (including WS-SecurityPolicy, WS-SecureConversation, WS-Trust), WS-Referral, and WS-Attachments and DIME (Direct Internet Message Encapsulation).

The .NET framework natively supports primitive SOA characteristics through its runtime environment and development tools, as explained here.

**Service encapsulation.** Through the creation of independent assemblies and ASP.NET applications, the .NET framework supports the notion of partitioning application logic into atomic units. This promotes the componentization of solutions, which has been a milestone design quality of traditional distributed applications for some time. Through the introduction of Web services support, .NET assemblies can be composed and encapsulated through ASP.NET Web Services. Therefore, the creation of independent services via .NET supports the service encapsulation required by primitive SOA.

**Loose coupling.** The .NET environment allows components to publish a public interface that can be discovered and accessed by potential clients. When used in conjunction with a messaging framework, such as the one provided by Microsoft Messaging Queue (MSMQ), a loosely coupled relationship between application units can be achieved. Further, the use of ASP.NET Web Services establishes service interfaces represented as WSDL

descriptions, supported by a SOAP messaging framework. This provides the foremost option for achieving loose coupling in support of SOA.

**Messaging.** When the .NET framework first was introduced, it essentially overhauled Microsoft's previous distributed platform known as the Distributed Internet Architecture (DNA). As part of both the DNA and .NET platforms, the MSMQ extension (and associated APIs) supports a messaging framework that allows for the exchange of messages between components. MSMQ messaging offers a proprietary alternative to the native SOAP messaging capabilities provided by the .NET framework. SOAP, however, is the primary messaging format used within contemporary .NET SOAs, as much of the ASP.NET environment and supporting .NET class libraries are centered around SOAP message communication and processing.

**Autonomy.** The .NET framework supports the creation of autonomous services to whatever extent the underlying logic permits it. When Web services are required to encapsulate application logic already residing in existing legacy COM components or assemblies designed as part of a traditional distributed solution, acquiring explicit functional boundaries and self-containment may be difficult. However, building autonomous ASP.NET Web Services is achieved more easily when creating a new service-oriented solution, as the supporting application logic can be designed to support autonomy requirements. Further, self-contained ASP.NET Web Services that do not share processing logic with other assemblies are naturally autonomous, as they are in complete control of their logic and immediate runtime environments.

**Reusability.** As with autonomy, reusability is a characteristic that is easier to achieve when designing the Web service application logic from the ground up. Encapsulating legacy logic or even exposing entire applications through a service interface can facilitate reuse to whatever extent the underlying logic permits it. Therefore, reuse can be built more easily into ASP.NET Web Services and any supporting assemblies when developing services as part of newer solutions.

**Stateliness.** ASP.NET Web Services are stateless by default, but it is possible to create stateful variations. By setting an attribute on the service operation (referred to as the WebMethod) called EnableSession, the ASP.NET worker process creates an HttpSessionState object when that operation is invoked. State management therefore is permitted, and it is up to the service designer to use the session object only when necessary so that statelessness is continually emphasized.

**Discoverability.** Making services more discoverable is achieved through proper service endpoint design. Because WSDL definitions can be customized and used as the starting point of an ASP.NET Web Service, discoverability can be addressed, as follows:

- The programmatic discovery of service descriptions and XSD schemas is supported through the classes that reside in the System.Web.Services.Discovery namespace. The .NET framework also provides a separate UDDI SDK.
- .NET allows for a separate metadata pointer file to be published alongside Web services, based on the proprietary DISCO file format. This approach to discovery is further supported via the Disco.exe command line tool, typically used for locating and discovering services within a server environment.
- A UDDI Services extension is offered on newer releases of the Windows Server product, allowing for the creation of private registries.
- Also worth noting is that Visual Studio contains built-in UDDI support used primarily when adding services to development projects.

**Open standards.** The .NET Class Library that comprises a great deal of the .NET framework provides a number of namespaces containing collections of classes that support industry standard, first-generation Web services specifications.

**Vendor diversity.** Because ASP.NET Web Services are created to conform to industry standards, their use supports vendor diversity on an enterprise level. Other non-.NET SOAs can be built around a .NET SOA, and interoperability will still be a reality as long as all exposed Web services comply to common standards (as dictated by the Basic Profile, for example). The .NET framework provides limited vendor diversity with regard to its development or implementation. This is because it is a proprietary technology that belongs to a single vendor (Microsoft). However, a third-party marketplace exists, providing numerous add-on products. Additionally, several server product vendors support the deployment and hosting of .NET Web Services and assemblies.

**Interoperable.** Version 2.0 of the .NET framework, along with Visual Studio 2005, provides native support for the WS-I Basic Profile. This means that Web services developed using Visual Studio 2005 are Basic Profile compliant by default. (Previous versions of Visual Studio can be used to develop Basic Profile compliant Web

services, but they require the use of third-party testing tools to ensure compliance.) Additional design efforts to increase generic interoperability also can be implemented using standard .NET first-generation Web services features.

**Federation.** Although technically not part of the .NET framework, the BizTalk server platform can be considered an extension used to achieve a level of federation across disparate enterprise environments. It supplies a series of native adapters and is further supplemented by a third-party adapter marketplace. BizTalk also provides an orchestration engine with import and export support for BPEL process definitions.

**Composable.** The .NET Class Library is an example of a composable programming model, as classes provided are functionally granular. Therefore, only those functions actually required by a Web service are imported by and used within its underlying business logic. With regard to providing support for composable Web specifications, the WSE supplies its own associated class library, allowing only those parts required of the WSE (and corresponding WS-\* specifications) to be pulled into service-oriented solutions.

**Extensibility.** ASP.NET Web Services subjected to design standards and related best practices will benefit from providing extensible service interfaces and extensible application logic (implemented via assemblies with service-oriented class designs). Therefore, extensibility is not a direct feature of the .NET framework, but more a common sense design approach to utilizing .NET technology. Functional extensibility also can be achieved by extending .NET SOAs through compliant platform products, such as the aforementioned BizTalk server.

**Service-oriented business modeling** concepts can be implemented with .NET by creating the standard application, entity-centric, and task-centric service layers. The orchestration layer requires the use of an orchestration engine, capable of executing the process definition that centralizes business workflow logic. Orchestration features are not a native part of the .NET framework. However, they can be implemented by extending a .NET solution environment with the BizTalk server platform.

**Logic level abstraction.** .NET SOAs can position ASP.NET Web Services and service layers to abstract logic on different levels. Legacy and net-new application logic can be encapsulated and wholly abstracted through proper service interface design. Service compositions can be built to an extent through the use of custom SOAP headers and correlation identifiers or by taking advantage of WSE extensions, such as the support provided for WS-Addressing and WS-Referral. (WSE also provides fundamental support for message-level security through extensions that implement portions of the WS-Security framework.) Because the .NET framework supports the development of industry-standard Web services, the proper positioning and application of service layers allows for the creation of the required layers of abstraction that promote fundamental agility. The use of an orchestration layer can further increase corporate responsiveness by alleviating services from business process-specific logic and reducing the need for task-centric services.

### **3.3.19 Web Services with Ajax and PHP**

#### **3.3.19.1 Reinventing the Web**

Back in 1996, the Web was incredibly exciting, but not a whole lot was actually happening on web pages. Programming a web page in 1996 often meant working with a static page, and maybe a bit of scripting helped manage a form on that page. That scripting usually came in the form of a Perl or C Common Gateway Interface (CGI) script, and it handled basic things such as authorization, page counters, search queries, and advertising. The most dynamic features on the pages were the updating of a counter or time of day, or the changing of an advertising banner when a page reloaded. Applets were briefly the rage for supplying a little chrome to your site, or maybe some animated GIF images to break the monotony of text on the page. Thinking back now, the Web at that time was really a boring place to surf. But look at what we had to use back then. HTML 2.0 was the standard, with HTML 3.2 right around the corner. You pretty much had to develop for Internet Explorer 3.0 or Netscape Navigator 2.1. You were lucky if someone was browsing with a resolution of 800 x 600, as 640 x 480 was still the norm. It was a challenging time to make anything that felt truly cool or creative. Since then, tools, standards, hardware technology, and browsers have changed so much that it is difficult to draw a comparison between what the Web was then and what it is today. Ajax's emergence signals the reinvention of the Web, and we should take a look at just how much has changed.

**Web Page Components.** When a carpenter goes to work every day, he takes all of his work tools: hammer, saw, screwdrivers, tape measure, and more. Those tools, though, are not what makes a house. What makes a house are

the materials that go into it: concrete for a foundation; wood and nails for framing; brick, stone, vinyl, or wood for the exterior—you get the idea. When we talk about web tools, we are interested in the materials that make up the web pages, web sites, and web applications, not necessarily the tools that are used to build them. Those discussions are best left for other books that can focus more tightly on individual tools. Here, we want to take a closer look at these web tools (the components or materials, if you will), and see how these components have changed over the history of the Web—especially since the introduction of Ajax.

**Classic Web Components.** The tools of the classic web page are really more like the wood-framed solid or wattle walls of the Neolithic period. They were crude and simple, serving their purpose but leaving much to be desired. They were a renaissance, though. Man no longer lived the lifestyle of a nomad following a herd, and instead built permanent settlements to support hunting and farming. In much the same way, the birth of the Web and these classic web pages was a renaissance, giving people communication tools they never had before.

The tools of the classic Web were few and simple:

- HyperText Markup Language (HTML)
- HyperText Transfer Protocol (HTTP)

Eventually, other things went into the building of a web page, such as CGI scripting and possibly even a database.

HTML provided everything in a web page in the classic environment. There was no separation of presentation from structure; JavaScript was in its infancy at best, and could not be used to create "dynamic HTML" through Document Object Model (DOM) manipulation, because there was no DOM. If the client and the server were to communicate, they did so using very basic HTTP GET and, sometimes, POST calls.

**Ajax.** Many more parts go into web sites and web applications today. Ajax is like the materials that go into making a high-rise building. High rises are made of steel instead of wood, and their exteriors are modern and flashy with metals and special glass. The basic structure is still there, though; walls run parallel and perpendicular to one another at 90-degree angles, and all of the structure's basic elements, including plumbing, electricity, and lighting, are the same—they are just enhanced.

In this way, the structure of an Ajax application is built on an underlying structure of XHTML, which was merely an extension of HTML, and so forth. Here are what I consider to be the tools used to build Ajax web applications:

- Extensible HyperText Markup Language (XHTML)
- Document Object Model (DOM)
- JavaScript
- Cascading Style Sheets (CSS)
- Extensible Markup Language (XML)

Now, obviously, other things can go into building an Ajax application, such as Extensible Stylesheet Language Transformation (XSLT), syndication feeds with RSS and Atom (of course), some sort of server-side scripting (which is often overlooked when discussing Ajax in general), and possibly a database.

XHTML is the structure of any Ajax application, and yes, HTML is too, but we aren't going to discuss older technology here. XHTML holds everything that is going to be displayed on the client browser, and everything else works off of it. The DOM is used to navigate all of the XHTML on the page. JavaScript has the most important role in an Ajax application. It is used to manipulate the DOM for the page, but more important, JavaScript creates all of the communication between client and server that makes Ajax what it is. CSS is used to affect the look of the page, and is manipulated dynamically through the DOM. Finally, XML is the protocol that is used to transfer data back and forth between clients and servers.

**Case study.** You may not think that changing and adding tools would have that much of an impact on how a site functions, but it certainly does. For a case study, I want to turn your attention to a site that actually existed in the classic web environment, and exists now as a changed Ajax web application. Then there will be no doubt as to just how far the Web has come. The following is a closer look at MapQuest, Inc. (<http://www.mapquest.com/>), how it functioned and existed in 2000, and how it functions today.

**Old application.** MapQuest was launched on 1996, delivering maps and directions based on user-defined search queries. It has been the primary source for directions and maps on the Web for millions of people ever since.

As MapQuest evolved, it began to offer more services than just maps and driving directions. By 2000, it offered traffic reports, travel guides, and Yellow and White Pages as well. How did it deliver all of these services? The same way all other Internet sites did at the time: click on a link or search button, and you were taken to a new page that had to be completely redrawn. The same held true for all of the map navigation. A change in the zoom factor or a move in any direction yielded a round trip to the server that, upon return, caused the whole page to refresh.

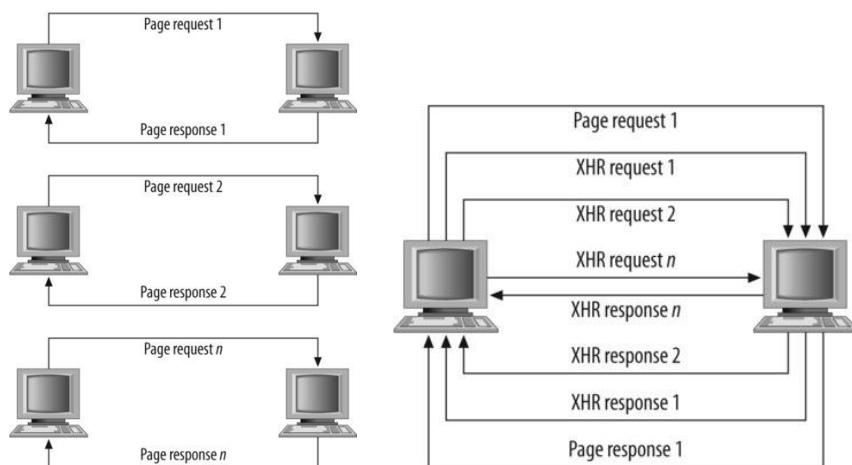
What you really need to note about MapQuest—and all web sites in general at the time—is that for every user request for data, the client would need to make a round trip to the server to get information. The client would query the server, and when the server returned with an answer, it was in the form of a completely new page that needed to be loaded into the browser. Now, this can be an extremely frustrating process, especially when navigating a map or slightly changing query parameters for a driving directions search. And no knock at MapQuest is intended here. After all, this was how everything was done on the Internet back then; it was the only way to do things.

The Web was still in its click-wait-click-wait stage, and nothing about a web page was in any way dynamic. Every user interaction required a complete page reload, accompanied by the momentary "flash" as the page began the reloading process. It could take a long time for these pages to reload in the browser—everything on the page had to be loaded again. This includes all of the background loading of CSS and JavaScript, as well as images and objects.

**New application.** In 2005, when Google announced its version of Internet mapping, Google Maps, everything changed both for the mapping industry and for the web development industry in general. The funny thing was that Google was not using any fancy new technology to create its application. Instead, it was drawing on tools that had been around for some time: (X)HTML, JavaScript, and XML. Soon after, all of the major Internet mapping sites had to upgrade, and had to implement all the cool features that Google Maps had, or they would not be able to compete in the long term.

Now, when you're browsing a map, the only thing on the page that refreshes when new data is requested is the map itself. It is dynamic. This is also the case when you get driving directions and wish to add another stop to your route. The whole page does not refresh, only the map does, and the new directions are added to the list. The result is a more interactive user experience.

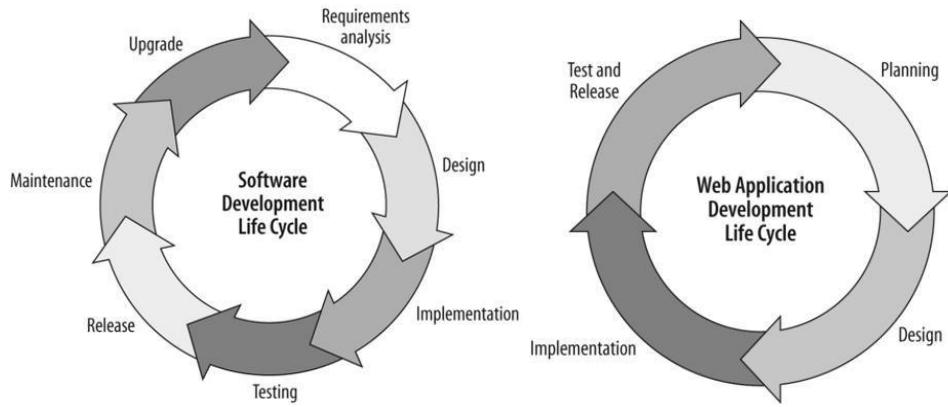
Ajax web applications remove the click-wait-click-wait scenario that has plagued the Web for so long. Now, when you request information, you may still perform other tasks on the page while your request (not the whole page) loads. All of this is done by using the Ajax tools discussed earlier, in the "Ajax" section of this chapter, and the standards that apply to them. After reading the section "Standards Compliance," later in this chapter, you will have a better idea of why coding to standards is important, and what it means when a site does not validate correctly (MapQuest, incidentally, does not). Figure 3.78 shows how Ajax has changed the flow of interaction on a web page.



**Fig. 3.78. (a) The flow of a typical interaction on the Web in 2000 (b) The flow of an Ajax interaction within a web page**

The addition of Ajax as a tool to use in web applications allows a developer to make user interaction more similar to that of a desktop application. Flickering as a page is loaded after user interaction goes away. The user will perceive everything about the web application as being self-contained. With this technology a savvy developer can make an application function in virtually the same way, whether on the Web or on the desktop.

**The transition.** The art of computer science slowly begins to creep back into the Web as the application life cycle begins. Any software developer can describe the life cycle of a software application. Following are the phases of software application development, it may be a little formal for most Ajax web development. Figure 3.79 shows a simpler Ajax web application life cycle. A lot of applications on the Web are the product of a more rapid development process, and simplifying the model makes it easier to keep that quick pace. An Ajax application can also follow the traditional development life cycle, but the simpler cycle fits better with the rapid iteration development style often used for web work.



*Fig. 3.79. (a) The typical software development life cycle (b) The Ajax web application development life cycle*

### 3.3.19.2 Ajax and Web Services

We've covered the basics, and now you know everything you need to get started using web services. It was the norm awhile ago to call a web service from a server script; the script would collect the data from the service, do its thing, and then send a new page to the client. It worked, at least as far as the user was concerned, as any other page on the Web did, so there was no way for him to know a web service was involved with this process.

Just like everything else, though, Ajax brings about new and fresh ways to look at existing technologies. Data gathered from a web service can now be placed on a page without an entire page reload, as you already know. The "wow" can be put into using web services, and they have a real place in web applications (and in particular, Ajax applications) today.

**Client requests.** For the most part, any client request for a web service is handled in one of two ways. Requests are made using a hidden `<iframe>` or `<frame>` element to handle the sending and receiving, and then the data is collected from the frames. Or, a call is made to a server script that handles the sending and receiving, and the client gets the data from the server-side script's response.

Why is this? Simply because of the (necessary) limitations placed on JavaScript with respect to calling pages on different domains. This is our sandbox, which we have seen before in this book. The two methods are simply ways of handling any restrictions a client may have.

There are exceptions, of course—in this case, the exception comes when the web services being contacted reside on the same domain as the page being called.

Up to this point, almost all of our examples of client requests to the server have fit a RESTful-like pattern. Unless there is some huge need for a different method, passing parameters to the server to define action and state is an easy way to implement a web service request. What we need to worry about is getting the data from a web service not located on our domain. The same domain communication will look the same, the server handles all of the requests to other servers, or it is the web service—either way, the request will look the same. Example 3.116 shows what the request will generally look like for all calls for a web service.

**Example 3.116. The client request for a web service**

```
/* The client request for a web service. */
new Ajax.Request('amazon.php', {
    method: 'get',
    parameters: { asin: '0596528388' },
    onSuccess: distributeResults});
```

Once the server sends back the data—in a format that the client will be expecting, no matter where it is from—it will parse and display the results in whatever manner is necessary.

**Service-side scripting to services.** Using PHP to access web services is fairly simple. The one detail that is necessary to address is whether the web service interfaces with SOAP or REST. This will determine what our server script will look like.

Think of this server-side script as an intermediary between the client and the web service. The client does not have to speak the web service's language, and in turn the web service does not have to speak the client's language. Our server script will handle all of the details. The advantage to this is that the intermediary takes care of all the parsing details instead of the client, which should offer some speed improvements.

PHP has a built-in class extension to the language to handle SOAP. Example 3.117 shows how this works.

**Example 3.117. A SOAP request to Amazon Web Service (AWS) using PHP**

```
<?php
/* A SOAP request to AWS using PHP.
 * This example shows how to create a SOAP request using PHP's SOAP extension to
 * an AWS method.*/
/* Create a new instance of the SOAP client class */
$client = new SoapClient('http://soap.amazon.com/schemas2/AmazonWebServices.wsdl');
/* Create the parameters that should be passed */
$params = Array(
    'asin'      => mysql_real_escape_string($_REQUEST['asin']),
    'type'      => 'lite',
    'tag'       => '[associates id]',
    'devtag'    => '[developer token]'
);
/* Call the AWS method */
$result = $client->ASINSearchRequest($params);
?>
```

SOAP functions are capable of returning one or multiple values. When there is only one value, the return value of the method will be a simple variable type. If multiple values are returned, however, the method will return an associative array of named output parameters.

Example 3.117 calls the ASINSearchRequest( ) method that is available with AWS. Other methods are also available, and each provides different search capabilities should the developer need them. Table 3.23 lists all of the methods available with AWS.

**Table 3.23. Methods available with AWS**

Method	Description
ASINSearchRequest( )	Performs an Amazon product code search and returns detailed information on the product.
BrowseNodeSearchRequest()	Performs a node search and returns a list of catalog items attached to the node.
KeywordSearchRequest( )	Performs a keyword search and returns the resulting products.
PowerSearchRequest( )	Performs an advanced search and returns the resulting products.
SellerSearchRequest( )	Performs a search for products listed by third-party sellers and returns the resulting products.
SimilaritySearchRequest()	Performs a search for items similar to a particular product code and returns the resulting products.

The following is an example of what the SOAP request will return:

```

Array
(
[Details] => Array
(
[0] => Array
(
[Url] => http://www.amazon.com/gp/product/0596528388%3ftag=[associates id]
%26link_code=xm%26camp=2025%26dev-t=[developer token]
[Asin] => 0596528388
[ProductName] => Ajax: The Definitive Guide
[Catalog] => Book
[Authors] => Array
(
[0] => Anthony T. Holdener III
)
[ReleaseDate] => 15 January, 2008
[Manufacturer] => O'Reilly Media
[ImageUrlSmall] => http://images.amazon.com/images/P/0596528388.01.THUMBZZZ.jpg
[ImageUrlMedium] => http://images.amazon.com/images/P/0596528388.01.MZZZZZZZ.jpg
[ImageUrlLarge] => http://images.amazon.com/images/P/0596528388.01.LZZZZZZZ.jpg
[ListPrice] => $49.99
[OurPrice] => $32.99
[UsedPrice] => $24.50
[Availability] => Usually ships within 24 hours
)
)
)

```

Then there is the REST way of handling things. Fortunately for us, AWS supports both SOAP and REST, so if I do not like the SOAP way of using the web service, I can use the REST methods. First, an XML Link must be created to pass to the AWS service. The structure of the XML Link is:

```
http://xml.amazon.com/onca/xml3?t=[associates id]&dev-t=[developer token]&
[Search Type]=[Search Term]&mode=books&sort=[Sort]&offer=All&type=[Type]&
page=[Page Number]&f=xml
```

Table 3.24 lists the parameters needed in the XML Link for it to be complete as far as AWS is concerned. Amazon has available for use with its web services is the ability to create the XML Link using its XML Scratch Pad, found at <http://www.amazon.com/gp/browse.html?node=3427431>.

**Table 3.24. Available XML Link options**

Option	Description
Search type	The type of search to perform: <ul style="list-style-type: none"> <li>• AsinSearch: Search for a single product using the ASIN.</li> <li>• AuthorSearch: Search for books by author.</li> <li>• BrowseNodeSearch: Search for products by BrowseNode category.</li> <li>• KeywordSearch: Search for products by keyword.</li> </ul>
Search term	This is dependent on the search type, and should correspond to it. For example, an AsinSearch should have a search term that is an Amazon product's ASIN.
Sort	The sort to be used on the search: <ul style="list-style-type: none"> <li>• +pmrank: Items are sorted by feature item.</li> <li>• +salesrank: Items are sorted by sales rank.</li> <li>• +reviewrank: Items are sorted by customer ratings.</li> </ul>
Type	The type of search results to display: <ul style="list-style-type: none"> <li>• Lite: Only essential product information is returned.</li> <li>• Heavy: All available product information is returned.</li> </ul>
Page number	The page number of the search results to jump to.

The data from a request of this nature will be an XML document with the following structure:

```
<Details url="http://www.amazon.com/gp/product/0596528388%3ftag=[associates id]
%26link_code=xm2%26camp=2025%26dev-t=[developer token]">
  <Asin>0596528388</Asin>
  <ProductName>Ajax: The Definitive Guide</ProductName>
  <Catalog>Book</Catalog>
  <Authors>
    <Author>Anthony T. Holdener III</Author>
  </Authors>
  <ReleaseDate>1 January, 2008</ReleaseDate>
  <Manufacturer>O'Reilly Media</Manufacturer>
  <ImageUrlSmall>
    http://images.amazon.com/images/P/0596528388.01.THUMBZZZ.jpg
  </ImageUrlSmall>
  <ImageUrlMedium>
    http://images.amazon.com/images/P/0596528388.01.MZZZZZZZ.jpg
  </ImageUrlMedium>
  <ImageUrlLarge>
    http://images.amazon.com/images/P/0596528388.01.LZZZZZZZ.jpg
  </ImageUrlLarge>
  <ListPrice>$49.99</ListPrice>
  <OurPrice>$31.49</OurPrice>
  <Availability>Usually ships in 24 hours</Availability>
  <UsedPrice>$24.50</UsedPrice>
</Details>
```

We've seen how to parse returned XML using PHP before, and Example 3.118 shows this.

#### **Example 3.118. A REST request to AWS using PHP**

```
<?php
/* A REST request to AWS using PHP.
 * This example shows how to create a REST request using PHP to an AWS method.
 */
$assoc_id = '[associate id]';
$dev_token = '[developer token]';
$xml_link = sprintf('http://xml.amazon.com/onca/xml3?t=%s&dev-t=%s&AsinSearch=
%s&mode=books&type=lite&f=xml',
    $assoc_id,
    $dev_token,
    urlencode($_REQUEST['asin'])
);
$results = file_get_contents($xml_link);
?>
```

**Gathering the Data.** Once the server has captured the data, whether the request was from SOAP or REST, it should be formatted in such a way that the client can parse it quickly. Using Example AJA.E8 as the model for getting the data from the server, all we must do is modify the last part to create an XML document the client will use. Example 3.119 shows what this looks like.

#### **Example 3.119. Gathering the AWS response and formatting it for the client**

```
<?php
/* Example. Gathering the AWS response and formatting it for the client.
 * This example shows how to create a REST request using PHP to an AWS method,
 * and how to parse and format the results. */

$assoc_id = '[associate id]';
$dev_token = '[developer token]';
$xml_link = sprintf('http://xml.amazon.com/onca/xml3?t=%s&dev-t=%s
    .&AsinSearch=%s&mode=books&type=lite&f=xml',
    $assoc_id,
    $dev_token,
```

```

urlencode($_REQUEST['asin'])
};

$results = file_get_contents($xml_link);
$xml = new SimpleXMLElement($results);
/* Was there a problem with the search query? */
if ($xml->faultstring) {
    $response = '<response code="500">' . $xml->faultstring . '</response>';
} else {
    $response = '<response code="200">';
    $response .= '<title>';
    $response .= '<name>' . $xml->details[0]->ProductName . '</name>';
    $response .= '<author>';
    $authors = '';
    /* Loop through authors and concatenate any names */
    foreach ($xml->details[0]->authors->author as $author) {
        /* Should a comma be added? */
        if (strlen($authors))
            $authors .= ', ';
        $authors .= $author;
    }
    $response .= $authors . '</author>';
    $response .= '<date>' . date('F j, Y',
        strtotime($xml->details[0]->ReleaseDate)) . '</date>';
    $response .= '<publisher>' . $xml->details[0]->Manufacturer . '</publisher>';
    $response .= '<img_src>' . $xml->details[0]->ImageUrlSmall . '</img_src>';
    $response .= '<availability>' . $xml->details[0]->Availability . '</availability>';
    $response .= '<list_price>' . $xml->details[0]->ListPrice . '</list_price>';
    $response .= '<amazon_price>' . $xml->details[0]->OurPrice . '</amazon_price>';
    $response .= '</title>';
    $response .= '</response>';
}
/* Change the header to text/xml so that the client can use the return string
 * as XML */
header('Content-Type: text/xml');
/* Give the client the XML */
print($response);
?>

```

**Sending the Web Service Response.** We have the data formatted the way we want the server to get it, and it should look something like this:

```

<response code="200">
    <title>
        <name>Ajax: The Definitive Guide</name>
        <author>Anthony T. Holdener III</author>
        <date>January 1, 2008</date>
        <publisher>O'Reilly Media</publisher>
        <img_src>
            http://images.amazon.com/images/P/0596528388.01.THUMBZZZ.jpg
        </img_src>
        <availability>Usually ships in 24 hours</availability>
        <list_price>$49.99</list_price>
        <amazon_price>$31.49</amazon_price>
    </title>
</response>

```

Notice that the `<response>` element has a `code` attribute attached to it. I am using HTTP status codes to show the status of the request, because passing these codes to the client will help it determine what to show the user. Now that the data has been sent to the client, it can be parsed easily and the results can be displayed to the user.

### **3.3.20 Web 2.0 Technologies and Web Services**

#### **3.3.20.1 Syndication**

The type of syndication in which sections of a web site are made available for other sites to use, most often using XML as the transport agent. News, weather, and blog web sites have always been the most common sources for syndication, but there is no limitation as to where a feed can come from.

**RSS** is not a single standard, but a family of standards, all using XML for their base structure. Note that I use the term standard loosely here, as RSS is not actually a standard. (RDF, the basis of RSS 1.0, is a W3C standard.) This family of standards for syndication feeds has a sordid history, with the different versions having been created through code forks and disagreements among developers. For the sake of simplicity, the only version of RSS that we will use in this book is RSS 2.0, a simple example of which you can see in Example 3.120.

##### **Example 3.120. A modified RSS 2.0**

```
<?xml version="1.0"?>
<rss version="2.0">
    <channel>
        <title>O'Reilly News/Articles</title>
        <link>http://www.oreilly.com/</link>
        <description>O'Reilly's News/Articles</description>
        <copyright>Copyright O'Reilly Media, Inc.</copyright>
        <language>en-US</language>
        <docs>http://blogs.law.harvard.edu/tech/rss</docs>
        <item>
            <title>Buy Two Books, Get the Third Free!</title>
            <link>http://www.oreilly.com/store</link>
            <guid>http://www.oreilly.com/store</guid>
            <description><![CDATA[ (description edited for display purposes...)]]>
            </description>
            <author>webmaster@oreillynet.com (O'Reilly Media, Inc.)</author>
            <dc:date></dc:date>
        </item>
        <item>
            <title>New! O'Reilly Photography Learning Center</title>
            <link>http://digitalmedia.oreilly.com/learningcenter/</link>
            <guid>http://digitalmedia.oreilly.com/learningcenter/</guid>
            <description><![CDATA[ (description edited for display purposes...)]]>
            </description>
            <author>webmaster@oreillynet.com (O'Reilly Media, Inc.)</author>
            <dc:date></dc:date>
        </item>
    </channel>
</rss>
```

Because of all the different versions of RSS and resulting issues and confusion, another group began working on a new syndication specification, called **Atom**. In July 2005, the IETF accepted Atom 1.0 as a proposed standard. There are several major differences between Atom 1.0 and RSS 2.0. Atom 1.0 is within an XML namespace, has a registered MIME type, includes an XML schema, and undergoes a standardization process. By contrast, RSS 2.0 is not within a namespace, is often sent as application/rss+xml but has no registered MIME type, does not have an XML schema, and is not standardized, nor can it be modified, as per its copyright.

##### **Example 3.121. A modified Atom feed**

```
<?xml version="1.0" encoding="utf-8"?>
<feed xmlns="http://www.w3.org/2005/Atom" xml:lang="en-US">
    <title>O'Reilly News/Articles</title>
    <link rel="alternate" type="text/html" href="http://www.oreilly.com/" />
    <subtitle type="text">O'Reilly's News/Articles</subtitle>
    <rights>Copyright O'Reilly Media, Inc.</rights>
    <id>http://www.oreilly.com/</id>
```

```

<updated></updated>
<entry>
  <title>Buy Two Books, Get the Third Free!</title>
  <id>http://www.oreilly.com/store</id>
  <link rel="alternate" href="http://www.oreilly.com/store"/>
  <summary type="html">  </summary>
  <author>
    <name>O'Reilly Media, Inc.</name>
  </author>
  <updated></updated>
</entry>
<entry>
  <title>New! O'Reilly Photography Learning Center</title>
  <id>http://digitalmedia.oreilly.com/learningcenter/</id>
  <link rel="alternate"
href="http://digitalmedia.oreilly.com/learningcenter/">
  <summary type="html">  </summary>
  <author>
    <name>O'Reilly Media, Inc.</name>
  </author>
  <updated></updated>
</entry>
</feed>

```

**XSLT** is an XML-based language used to transform, or format, XML documents. On 1999, XSLT version 1.0 became a W3C Recommendation. As of 2007, XSLT version 2.0 is a Recommendation that works in conjunction with XPath 2.0. XSLT uses XPath to identify subsets of the XML document tree and to perform calculations on queries. XSLT takes an XML document and creates a new document with all of the transformations, leaving the original XML document intact. In Ajax contexts, the transformation usually produces XHTML with CSS linked to it so that the user can view the data in his browser.

Another way to access data is through a web feed. **Web feeds** are also XML-based documents that are structured with either RSS or Atom. Feeds are generated in a number of ways, and are not as "flashy" as web services because they are static in nature.

One way you can make a feed is by writing a script that visits pages and scrapes data from them. **Web scraping** is not the noblest way to get data, and it could break laws if you take copyrighted material from a site and use it on yours without proper authorization. Laws vary by state and country, but I recommend avoiding this practice and using more readily available information through syndicated feeds.

Using a syndicated web feed to gather information is a good way to create your own web services to use on your site. Feeds can be pulled down by your server script and then disseminated to clients when the information is requested. This is a good solution for quickly collecting headlines and descriptions from a source that might not provide web services to grab information. The only big disadvantage to using syndicated resources is that you are at the mercy of the provider. Whatever content the provider wishes to place in a feed is what you will get; there is no choice on the developer's part.

What is good about feeds, however, is that you can find feed aggregators to do some of the work for you. Aggregators basically gather feeds from content providers and aggregate them in a number of ways to make it easier to get data for your application. A good example of this is Google Reader, found at <http://www.google.com/reader>. This application takes feeds from different sources (in real time) and allows subscribers to customize their choices. Using feeds, you keep your application completely up-to-date.

**RSS and Atom.** There is nothing new about syndication, RSS, or Atom unless you create a new web service that aggregates feeds and sends them to the client. This creates a service that distributes the information you want without you having to be the keeper of the data (and dealing with all of the hassle that goes along with that).

For example, consider this feed from Yahoo! Weather (<http://weather.yahooapis.com/forecastrss?p=62221>):

```

<?xml version="1.0" encoding="UTF-8" standalone="yes" ?>
<rss version="2.0" xmlns:yweather="http://xml.weather.yahoo.com/ns/rss/1.0">

```

```

xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#">
<channel>
    <title>Yahoo! Weather - Belleville, IL</title>
    <link>
        http://us.rd.yahoo.com/dailynews/rss/weather/Belleville_IL/*
        http://weather.yahoo.com/forecast/62221_f.html
    </link>
    <description>Yahoo! Weather for Belleville, IL</description>
    <language>en-us</language>
    <lastBuildDate>Tue, 13 Mar 2007 11:55 am CDT</lastBuildDate>
    <ttl>60</ttl>
    <yweather:location city="Belleville" region="IL" country="US" />
    <yweather:units temperature="F" distance="mi" pressure="in" speed="mph" />
    <yweather:wind chill="77" direction="210" speed="14" />
    <yweather:atmosphere humidity="50" visibility="1609"
        pressure="30.07" rising="2" />
    <yweather:astronomy sunrise="7:15 am" sunset="7:05 pm" />
    <image>
        <title>Yahoo! Weather</title>
        <width>142</width>
        <height>18</height>
        <link>http://weather.yahoo.com/</link>
        <url>http://l.yimg.com/us.yimg.com/i/us/nws/th/main_142b.gif</url>
    </image>
    <item>
        <title>Conditions for Belleville, IL at 11:55 am CDT</title>
        <geo:lat>38.5</geo:lat>
        <geo:long>-90</geo:long>
        <link>
            http://us.rd.yahoo.com/dailynews/rss/weather/Belleville_IL/*
            http://weather.yahoo.com/forecast/62221_f.html
        </link>
        <pubDate>Tue, 13 Mar 2007 11:55 am CDT</pubDate>
        <yweather:condition text="Fair" code="34" temp="77"
            date="Tue, 13 Mar 2007 11:55 am CDT" />
        <description>
            <![CDATA[
                <br />
                <b>Current Conditions:</b><br />
                Fair, 77 F<BR /><BR />
                <b>Forecast:</b><br />
                Tue - Partly Cloudy. High: 81 Low: 58<br />
                Wed - Partly Cloudy. High: 76 Low: 55<br />
                <br />
                <a href="http://us.rd.yahoo.com/dailynews/rss/weather/
                    Belleville_IL/*
                    http://weather.yahoo.com/forecast/62221_f.html">
                    Full Forecast at Yahoo! Weather
                </a><BR />
                (provided by The Weather Channel)<br />
            ]]>
        </description>
        <yweather:forecast day="Tue" date="13 Mar 2007" low="58" high="81"
            text="Partly Cloudy" code="30" />
        <yweather:forecast day="Wed" date="14 Mar 2007" low="55" high="76"
            text="Partly Cloudy" code="30" />
        <guid isPermaLink="false">62221_2007_03_13_11_55_CDT</guid>
    </item>
</channel>
</rss>
```

A little bit of PHP code can turn this feed into a REST web service without any trouble. Example 3.122 shows how to do this.

#### **Example 3.122. Using feeds to distribute information with PHP**

```

<?php
/* Using feeds to distribute information with PHP.
 * This example shows how to take an RSS feed and strip out the information
 * desired by the application, giving only a minimal amount of data to the client.*/
/* Get the RSS feed from Yahoo! for my zip code */
$results = file_get_contents('http://weather.yahooapis.com/forecastrss?p=62221');
$xml = new SimpleXMLElement($results);
/* Create the XML for the client */
$response = '<response code="200">';
$response .= '<weather>';
/* Create the prefix context for the XPath query */
$xml->registerXPathNamespace('y', 'http://xml.weather.yahoo.com/ns/rss/1.0');
/* Gather the temperature data */
$temp = $xml->xpath('//y:condition');
/* Fill in information for the client */
$response .= '<temp>' . $temp[0]['text'] . ', ' . $temp[0]['temp'] . ' &#176;F</temp>';
$response .= '<img>' . $xml->channel->image->url . '</img>';
$response .= '<link>' . $xml->channel->image->link . '</link>';
$response .= '</weather>';
$response .= '</response>';
/* Change the header to text/xml so that the client can use the return string
 * as XML*/
header('Content-Type: text/xml');
/* Give the client the XML */
print($response);
?>

```

You could easily change this code to point to several different sources, aggregate that information, and then send it out for the client to display. The whole point of this is so that an Ajax call can be made—say, every 15 minutes—to update the application with the latest information.

**Feed Validation.** It is all well and good to collect information from different sources using RSS feeds, but how do you know you are getting what you expect? Feed validation services are available to check the validity of a feed so that you can be more certain that your application will work with it. A good idea is to periodically check a feed's validity to ensure that your aggregation service is working. Better yet, write an application that does the feed validation for you automatically. There are a number of feed validators to choose from; here are a few to get you started:

- **Feed Validator** (<http://feedvalidator.org/>). This validator is extremely versatile and can check RSS 0.90, 0.91, 0.92, 0.93, 0.94, 1.0, and 2.0 feeds as well as Atom feeds.
- **W3C Validator** (<http://validator.w3.org/feed/>). This is the W3C validator for checking the validity of RSS and Atom feeds. It works in the same manner as the W3C HTML/XHTML and CSS validators.
- **Redland RSS 1.0 Validator** (<http://librdf.org/rss/>). This validator from validates and formats results for display.
- **Experimental Online RSS 1.0 Validator** ([http://www.ldodds.com/rss\\_validator/1.0/validator.html](http://www.ldodds.com/rss_validator/1.0/validator.html)) This is a prototype validator based on a Schematron schema for validating RSS 1.0.
- **RSS Validator** (<http://rss.scripting.com/>). This is a feed validation service that tests the validity of RSS 0.9x feeds.

### 3.3.20.2 Mashups

Combining two or more web services can yield some functional and easy-to-use applications. On their own, each service may be good but harder to use, or less useful, than it would be if it were combined with other services. A mashup is what you get when you combine web services. The result is a Web 2.0 application that is more sophisticated than its parts and provides functionality that most likely did not exist before.

**Mashups in Web 2.0 Applications.** Mashups can aid in the development of Web 2.0 applications by giving them better interactivity with maps and associated data, and can aid in the manipulation of blogs, lists, photo and video sharing, and just about any other type of service found on the Web. Without the capabilities these mashups provide, some of these existing applications would lack the necessary functionality to truly be considered Web 2.0 applications. Even worse, they may not be as useful as they could be. Web services in general, but especially

when they are combined into mashups, help to give web applications a dynamic and often flashy appearance. This, in turn, makes users feel like they are using an application, and not just viewing a web site. Mashup creation ushered in the era of Web 2.0 applications and their underlying programming. Of course, Web 2.0 applications have since evolved into more than mashups, combining dynamic HTML with visual effects and better user interaction.

**What Are Mashups?** A simple definition of a mashup is a web site or application that combines two or more sources of information into a new web application. Another way to look at a mashup is to think of it as a hybrid web application, where parts of the application come from public interfaces such as syndicated web feeds, site scrapings, and web services. The word **mashup** first became popular in the music industry, as DJs from around the world began to combine parts of existing music tracks (sometimes of different genres) to create entirely new tracks. These mashups became hits in the club scene, as a lot of them were from the techno/dance genre. When web services started to be combined to form new applications for the Web, the term mashup easily transitioned to this new medium.

**A Brief History.** The first noticeable public web application that used two different APIs was launched in April 2005. This application was the (now famous) **HousingMaps.com**. By hacking the JavaScript that Google used for its maps and combining it with the classified site Craigslist, the author created a site that allows users to visually search houses in major U.S. cities. Soon, though, Google and other search engine companies released publicly available APIs to their resources that allowed the web development community to respond with hundreds of mashups.

**Mashups As Applications.** Most web services are more data-driven than anything else, and most likely will not overwhelm an application. This is due, in large part, to the level of control the developer has on this data. However, some web services could be distracting or overwhelming to an application if the developer is not careful. Mapping services are a good example of this type of service. Other mashups strategically use different web services to create a usable and welcomed application for the Web. These mashups do not overwhelm a user with information, unless she specifically asks for it. They can stand on their own, and they need no additions to make them more useful.

**What Mashups Can Do.** Mashups can do just about anything you want them to do. Because all mashups are really existing services that are combined to create a new service on the Web, the sky is the limit. I recommend that you think about what you need to accomplish, and determine whether others have already handled parts of that task. If so, determine whether you can leverage that work with what you are doing. Search for public information; chances are good that what you need is available. Not all publicly available information is free, mind you, but the amount of data you can find on the Web is amazing. Look for open source web services. That may require a little extra searching, but finding a web service that adds no costs to your application is worth the extra effort. You can tailor mashups to do anything you want. With a little bit of work, you can put together available web services to create brand-new functionality. Open source services will cut down on costs and development efforts, and publicly available data can effectively provide the data an application needs. So, you may be asking, "What can mashups not do?" Practically nothing. Sometimes there may be a better alternative, but overall, mashups can function in just about any situation you could think of.

**Data Sources.** A big part of what goes into a mashup is the data sources that are used to create the different components in the application. Plenty of fee-based services are on the Web, but unless the mashup supports an application for a large corporation the price tag can be unrealistic for most individuals and small companies to pay. For those groups of people, it is better to try to find free or open source services to provide the data for any application that is to be built. A lot of times, finding the data sources for an application can be harder than coding it. The availability of data is only as good as the services are at advertising it to the world. Even data that is publicly available, or is at least supposed to be in the public record, can be buried within pages and impossible to find. Also, a lot of publicly accessible data is available only for a price. Being able to factor these variables into the budget for a project can sometimes keep the project from being canceled completely. It is important to know what needs to go into your mashup.

**Public Data.** When it comes to public data, a wide variety of information is available, from demographics to death records. A lot of this information comes at a cost, though some government agencies are beginning to allow access to some information for free. The idea that it is public can excite many developers before they realize that the data comes at a price. Remember that there is a clear distinction between public and free—they are not the same thing. There is a lot out there for those who look for it; some examples of publicly available data are:

- Public records
- Background check records
- Business records
- People searches

**Public records.** Public data encompasses all the data we are trying to collect. I define public records as all the records available to anyone who walks into a county clerk's office and asks for them. Examples include records of births, deaths, marriages, divorces, and bankruptcies, as well as property records. You can access these records online, but unfortunately they come with a price tag. Sites exist that allow a user to search for data and then pay a fee to get the information. Some examples are:

- People Finders (<http://www.peoplefinders.com/>)
- Public Record Finder (<http://www.publicrecordfinder.com/>)

The caveat to public records is that not all states supply the needed information in a way that you can access it easily via the Web. For example, you can search birth records only for the states of California and Texas. Death records are provided only for people who possessed Social Security numbers. Marriage records can be searched in only about one-quarter of the USA's country's states, and the date ranges for licenses vary by state:

**Background check records.** Typically, people conduct background checks if they want to determine whether someone has a criminal or sex offender record. Thankfully, the federal government provides several sources for such information. It costs more to obtain background check records than public records, because they are more difficult to obtain. However, they can be worth the added cost because you can obtain almost everything you want to know about a person through a background check.

**Business records.** Data on individuals is not the only public data available. Data on businesses (both large and small) is also publicly available. You can learn everything you want to know about a business from web sites that perform searches for you. A typical business would have the following information available: its legal name, officers or owners, address, state and federal tax liens, filing information, DBA business name filings, and property ownership. This is good information to have and could greatly enhance mapping mashups. Professional licenses are also on record and are available for searching. You can generally find the name of the business, license owner, address, and other related information.

**People searches.** Finally, we have public data that is gathered and kept by corporations, mainly for marketing purposes. Most of the name or phone number searches performed on the Web are conducted from these marketing sources. Here are some of the common pieces of information marketing companies compile:

- Full name
- Age/date of birth
- Address
- Phone number
- Social Security number

It is frightening to realize that a marketing company can purchase such information for its databases. Some of this information is also available for anyone that owns a land-line phone number—phone books are online, and include the names, addresses, and phone numbers of individuals.

**Open Source Services.** Open source services are the way to go if you want to keep your mashups as inexpensive as possible. However, they may not provide the level of support that fee-based services provide. Nonetheless, open source services usually make it easy to get data that may not exist anywhere else on the Web, or data that was available but not easily accessible through an API. Finding these services can be easy enough, as certain web sites are solely dedicated to listing available web services on the Internet. The following is a list from 2008 of a few of the better sites that track web services:

- Web Service List (<http://www.webservicelist.com/>)
- WebserviceX.NET (<http://www.webservicex.net/WS/default.aspx>)
- Programmable Web (<http://www.programmableweb.com/>)
- Webmashup.com (<http://www.webmashup.com/>)

**Application Portlets.** Portlets are components that you can easily plug into applications and aggregate into a page. Public web services I can be integrated into applications to give a little Web 2.0 feel were basically portlets in the application. These web services (even when combined) did not really comprise a new mashup. Mashups are

created only when individual services are directly combined. An application composed of three or more web services or portlets that do not necessarily interact with one another should still be considered a mashup. The individual portlets comprise a mashup that is basically an information web portal. An example of mashup: use the following web services Flickr, Technorati, Yahoo! Image Search, and YouTube. Web portals usually are composed of individual portlets that contain information regarding a main theme. This theme makes the portal, but the inclusion of many web services makes it a mashup.

**Building a Mashup.** By following these four easy steps, you will be on your way to building your own unique mashups for the web world to consume:

1. Choose a subject.
2. Select data sources.
3. Decide on the backend environment and language.
4. Code it.

It is very hard to program anything without a little bit of direction. Start with the simplest question about the new program you are going to create: what is this a mashup of? Although it may be tempting to jump in with both feet and build a mashup that combines data sources from different areas—such as maps, real estate data, photos, search capability, and more—don't do it! By narrowing down the subject of your mashup, you also help to eliminate web services that you will not need while deciding which ones you do need. Now that you know roughly the types of web services you require for your mashup, it is time to hunt for them on the Internet.

The data sources you choose will directly affect the web services (and their APIs) that you picked. For example, you know you want to use the mapping data that Google provides, so you are going to use its API out of necessity. On the other hand, you may have found the data you were looking for from a government site, and you have to scrape the data yourself to make it usable in your mashup. In such cases, you have direct control over how to get the data, so you have more choices. I suggest that you choose APIs that have good documentation associated with them so that you will have an easier time programming with them. For sources that do not have APIs, it is best if the data is given in a straightforward manner that is easy to obtain. The way the data is presented to you must always be consistent (e.g., if it is given in a tabular manner, the columns should always be in the same order).

You need to make a couple of decisions regarding the backend of the mashup before you begin programming. The first (and most obvious) decision is the language you are going to use. It makes no difference whether you use PHP, C# .NET, Perl, or Java, as long as you know the language. Sometimes the API you are using works specifically with a certain language, but most times it will not matter. A big factor to consider with the backend is the transport type being used. You must make sure you know how to create connections to the API and handle the data coming back, regardless of whether you are using SOAP, REST, or XML-RPC.

**Mashups and Business.** It is extraordinary how mashups have taken off to such an extent that they are now a viable business solution. This is especially true when it comes to mapping services and the types of mashups you can create with them. This blending of existing technologies can have a great impact on a business when leveraged properly. You may not see this without a couple of "for examples," so I will provide them!

First, consider a delivery company and its need to know where all of its trucks are at any given time. A web application that blends mapping services with a custom-built GPS service could drastically change how the company does business, as an application of this sort could be used for a number of different things. It could alert a controller if any of the company's drivers was exceeding the speed limit while out on a route. There is the reduced cost of not getting driving violations and all of those associated fines, plus there is the added benefit of not giving the company a negative reputation because of trucks that are speeding or have been pulled over. It could also track the time spent at a stop, or any of the statistical data an analyst would want to improve performance with routes and delivery methods.

Consider another example of a real estate company that wants to improve its online presence and increase its revenues at the same time. Again, using a mapping service combined with a real estate service and possibly other local data services, an application could be built to meet the company's needs. Think of the possibilities for this company if it could provide census data along with real estate data. The ability for consumers to understand the demographics and density data in a neighborhood from the comfort of their own homes would be very beneficial. It would most likely increase the real estate company's revenue as a direct side effect, as the agents would spend less time showing houses to customers that do not suit them. The Web provides great opportunities for searching beforehand so that making purchases is easier and less painful.

### **3.3.21 Publicly Available Web Services**

Public available Web services have supplied developers with the APIs to their services to make it easier to use them. This is also helpful for getting "up and running" quickly, something that is important in today's world of rapid application deployment. It would be impossible, however, to go over all of the publicly available web services that exist today; new ones pop up all the time while old and obsolete ones disappear, and a service's interface could change over time to better reflect the needs of developers. Instead, I will highlight some of the better known, well-documented, and more readily used web services to give you a leg up on beginning to develop with them. And though they may change, at least you'll have the foundation available to begin programming.

#### **3.3.21.1 Blogging Services**

Blogging has become a huge phenomenon over the past several years, and blogs can be found in both the private and professional sectors. In fact, the line between personal and professional blogs, especially concerning technology, has blurred. After all, most of us cannot help but write about and experiment with technology even when it is on our own time.

Larger blog sites and operations have begun to see a real demand for services that allow developers to access the blogging system's content and controls programmatically from within other applications. This has also led to the pleasant side effect of third-party tools being developed for programmatic remote access to blog systems.

These APIs range from direct access of management tools for blogging software to simple searching of blog content, and everything in between. In this section, I will list some of the more popular APIs of these blogging web services and discuss their functionality so that you will know what blog tools are available to you.

The following is a list of some of the more popular blogging APIs in 2008:

**MSN Spaces** is a service that allows external editing of text and attributes of weblog posts using the MetaWeblog programming interface. You can find the API documentation for this interface at <http://msdn2.microsoft.com/en-us/library/bb259702.aspx>.

**Akismet** is a blogging tool to help prevent spam from interfering with a blog's content or usability. It is available for personal and commercial use, and you can find the API to interact with it programmatically at <http://akismet.com/development/api/>.

**TypePad** is a blog service catering to professionals and small businesses looking to create blogs for their sites. With its Atom API, you can do such things as programmatically post and read a blog. The API documentation is at [http://www.sixapart.com/pronet/docs/typepad\\_atom\\_api](http://www.sixapart.com/pronet/docs/typepad_atom_api).

**FeedBurner** is a library of web services focused on blogging, and offers common blog functionality programmatically through its two APIs: FeedFlare and Feed Awareness. You can find documentation on these APIs at <http://www.feedburner.com/fb/a/developers>.

**FeedBlitz** is a blogging service that allows you to manage a blog through email for services such as user profiles, subscriptions, and syndications. You can find the API documentation to manage this information programmatically at <http://feedblitz.blogspot.com/2006/10/feedblitz-api.html>.

**Weblogs.com** is a ping service used to automatically inform VeriSign whenever the content of your site's blog is updated so that it can, in turn, note this change on its web site. You can find the API documentation for this interaction at <http://weblogs.com/api.html>.

**Technorati** is a service that keeps track of the millions of blogs that are on the Web today, organizing them and other media forms to make them more manageable. You can find the API documentation to manage these services programmatically at <http://developers.technorati.com/wiki>.

A good example of using a blogging service API in an application is FeedBurner's Feed Management API, also known as MgmtAPI on the FeedBurner site. The MgmtAPI enables FeedBurner publishers to create and manage

their accounts programmatically and can facilitate the following functions: find, get, add, modify, delete, and resync. This capability can be useful on its own or combined with other web services to create a mashup. All of these blog APIs may not work in the same way, but they are all fairly easy to use.

For a simple example, we will look at the MgmtAPI Find Feeds functionality. MgmtAPI uses the REST protocol for all its functionality, and returns XML for its response. The Request URL looks like this:

```
http://api.feedburner.com/management/1.0/FindFeeds?user=[username] &
password=[password]
```

This method has no parameters associated with it, other than the standard authentication parameters, and it simply returns a list of all the feeds associated with the user.

The response XML has a schema document located at <http://api.feedburner.com/management/1.0/FindFeedsResponse.xsd>. A sample response for a Find Feeds request would look something like this:

```
<?xml version="1.0" encoding="utf-8" ?>
<feeds>
    <feed id="6" uri="trey-rants" title="Where Are the Standards?" />
    <feed id="26" uri="trey" title="Using OpenOffice as Your Word Processor" />
    <feed id="1999" uri="trey-sarah" title="Latest with the Twins" />
</feeds>
```

No server-specific errors were associated with this request. If the user passed to the server does not have any feeds, the feeds element in this simply has no children.

Example 3.123 shows the basics of how to call this request within PHP and handle the results. In this example we use the PEAR library HTTP:Request for better error handling of the response. You can do what you want with the response.

### ***Example 3.123. Calling the FeedBurner MgmtAPI's Find Feeds method using PHP***

```
<?php
/* Calling the FeedBurner MgmtAPI's Find Feeds method using PHP.
 * This file is just a simple example of fetching data using REST and parsing
 * the results. Much more could be done on the server side, especially if part
 * of a mashup, in this script.
 * The file, user.inc, contains the username and password for the REST request. */
require_once('user.inc');
/** This is the PEAR HTTP:Request library used to make our request. */
require_once('HTTP/Request.php');
/* Set up the request */
$request =& new HTTP_Request("http://api.feedburner.com/management/" .
    ."1.0/FindFeeds?user=$username&password=$password");
$request->addHeader('Accept', 'application/atom+xml');
/* Get the results of the request */
$results = $request->sendRequest();
/* Begin the response text */
$response = '<?xml version="1.0" encoding="utf-8"?>';
/* Was there a problem with the request? */
if (PEAR::isError($results))
    $response .= '<response code="500">' . $response->getMessage() . '</response>';
else {
    $code = $request->getResponseCode();
    $response .= '<response code="'.$code.'">';
    $xml = new SimpleXMLElement($results);
    /* We can check for whatever codes we want here */
    switch ($code) {
        case 200:
            /* Were there any feeds? */
            if ($xml->children()) {
                $response .= '<feeds>';
```

```

    /* Loop through the feeds */
    foreach ($xml->children() as $feed) {
        $response .= '<feed>';
        $response .= "<title>{$feed['title']}</title>";
        $response .= "<link>{$feed['uri']}</link>";
        $response .= '</feed>';
    }
    $response .= '</feeds>';
} else
    $response .= '<feeds />';
break;
default:
    $response .= '<feeds />';
break;
}
$response .= '</response>';
}
/* Change the header to text/xml so that the client can use the return string as XML
 */
header('Content-Type: text/xml');
/* Give the client the XML */
print($response);
?>

```

### 3.3.21.2 Bookmark Services

Bookmarking, at least in terms of the sites that offer it, is the ability to track and share (by category) saved bookmarks with others around the world. The del.icio.us service has made bookmarking popular, as the del.icio.us web site is by far one of the most popular when it comes to blogging, photo and video sharing, and so on.

Here are some of the well-known web service APIs that are available in 2008:

**del.icio.us** is a bookmarking service used to keep track of the types of material an individual may be interested in, and allows users to share these bookmarks with others. You can find the API documentation to programmatically control this functionality at <http://del.icio.us/help/api/>.

**Simpyle** is a bookmarking service for social interaction that enables users to tag and share bookmarks and notes. The web service allows for programmatically interfacing with the site. You can find the API documentation at <http://www.simpyle.com/doc/api/rest>.

**Blogmarks** is another social bookmarking service, though it is more "blog-like" than an actual blog. Through the use of its AtomAPI, you can retrieve Atom feeds from the site that GET, POST, DELETE, and PUT bookmarks. The AtomAPI documentation is at <http://dev.blogmarks.net/wiki/DeveloperDocs>.

**Ma.gnolia** allows developers to access features for managing and collecting bookmark data from their site from other applications. You can find the API documentation to interact with Ma.gnolia from outside programs at <http://ma.gnolia.com/support/api>.

These web service APIs are easy enough to use, but because of its popularity, I'll show a little more of the del.icio.us API. del.icio.us uses REST over HTTPS for all of its API calls, so a username and password are required just as they were in Example AJA.E11. The REST URI for adding a post to del.icio.us programmatically is:https://api.del.icio.us/v1/posts/add?

For a successful post, the server will respond as follows:

```
<result code="done" />
```

If the post failed, however, the response will be:

```
<result code="something went wrong" />
```

Using PHP on the server, Example 3.124 shows how an Add request can be executed with PEAR and the response captured to notify the client. This example assumes that the information to post is coming from a POST from the client.

#### **Example 3.124. Adding a post programmatically to del.icio.us using PHP**

```
<?php
/* Adding a post programmatically to del.icio.us using PHP.
 * This file demonstrates how to take the form post from the client and post
 * this data to del.icio.us using a REST architecture. The result from the
 * post is sent back to the client.
 * The file, user.inc, contains the username and password for the REST request.*/
require_once('user.inc');
/* This is the PEAR HTTP\Request library used to make our request. */
require_once('HTTP/Request.php');
/* Is there something to post? */
if (!isempty($_REQUEST['description']) && !isempty($_REQUEST['url'])) {
    /* Set up the request */
    $request =& new HTTP_Request("https://$username:$password@api.del.icio.us/v1"
        ."/posts/add?url=".$_REQUEST['url']. "&description="
        .$_REQUEST['description']);
    $request->addHeader('Accept', 'application/atom+xml');
    /* Get the results of the request */
    $results = $request->sendRequest();
    /* Begin the response text */
    $response = '<?xml version="1.0" encoding="utf-8"?>';
    /* Was there a problem with the request? */
    if (PEAR::isError($results))
        $response .= '<response code="500">'.$results->getMessage(). '</response>';
    else {
        $code = $request->getResponseCode();
        $response .= '<response code="'.$code.'">';
        $xml = new SimpleXMLElement($results);
        $response .= $xml['code'];
        $response .= '</response>';
    }
} else
    $response .= '<response code="500">There was nothing to post.</response>';
/* Change the header to text/xml so that the client can use the return string as XML
 */
header('Content-Type: text/xml');
/* Give the client the XML */
print($response);
?>
```

### **3.3.21.3 Financial Services**

Financial institutions are beginning to see the advantages of having a web service attached to their financial information. Most services are offered at a price, as these types of services are more useful to a corporation or business than to an individual. Examples of services being offered today are those that involve the various stock markets, accounting control to an application, invoicing, credit checking, mutual fund prices, and currency rates.

A sample of some of these web services follows:

**Blinksale** is an online invoice service that allows users to invoice clients from a web browser. The API makes it easier for a developer to build this functionality into another application, and you can find it at <http://www.blinksale.com/api>.

**StrikeIron Historical Stock Quotes** service provides developers with a means for gathering detailed information on a stock ticker symbol for a specified date programmatically in an application. The API documentation for this service is at <http://www.strikeiron.com/developers/default.aspx>.

**Dun and Bradstreet Credit Check** enables developers to do credit checks against potential business clients programmatically through SOAP requests. You can find the API documentation providing access to these credit checks at <http://www.strikeiron.com/ProductDetail.aspx?p=223>.

**NETaccounts** offers financial accounting from the web browser. Through its API it offers the basic CRUD operations. The API documentation showing these operations is available at <http://www.netaccounts.com.au/api.html>.

These services give you everything you need to start adding financial web service data into your own Ajax applications without much work. The only real downside with financial services is that they all charge fees for usage, but if you can justify those costs, the benefits of adding this kind of data can be enormous.

### 3.3.21.4 Mapping Services

One of the hottest topics for application development is in the realm of web mapping. This subject, more commonly referred to as Geographic Information Systems (GIS) within the industry, has some promising applications. This is especially true when used in a mashup with other available data sets.

Most people are visual by nature; as such, using maps to help convey a stat or fact adds good value to an application. For example, is it easier to get a list of addresses for restaurants in a district of a city you are unfamiliar with, or to get a map with the locations marked on it? Think about what started with MapQuest and driving directions, and is now a commonplace and expected function of these types of sites—a map marking the route with supplementary text giving step-by-step directions. This kind of visual aid is what web mapping is all about. Having access to mapping service APIs opens the door for developers to create many new and useful tools.

Many mapping services are available; here we will look at several of the bigger players so that you can get an idea of what they offer. A sample of the API, and of course, something visual, will accompany the list of mapping services.

**Google Maps** is a service that allows a developer to place maps in an application, with the use of JavaScript to control the map's features and functions. This API, which gives developers all the functionality found with Google Maps for their own applications, is available at <http://www.google.com/apis/maps/>.

**Yahoo! Maps** enables developers to publish maps, created from Yahoo!'s engine, on their sites. The API allows for easy integration into existing applications, quickly giving a site a Web 2.0 look. You can find the API documentation for all of the Yahoo! Maps functionality at <http://developer.yahoo.com/maps/>.

**ArcWeb**, a service provided by ESRI, allows you to integrate mapping functionality into a browser, without having to create it from scratch. You can find the API to use this service at <http://www.esri.com/software/arcwebservices/index.html>.

**FeedMap's BlogMap** allows a developer to geocode a blog, making it locatable by geographic area. The API documentation outlining this functionality is available at <http://www.feedmap.net/BlogMap/Services/>.

**Microsoft MapPoint** is a mapping service that enables you to integrate high-quality maps and GIS functionality into an existing web application with minimal effort. The API documentation for MapPoint is available at <http://msdn.microsoft.com/mappoint/mappointweb/default.aspx>.

**MapQuest's OpenAPI** lets developers use JavaScript to integrate maps into a web application or site. The API documentation for OpenAPI is located at [http://www.mapquest.com/features/main.adp?page=developer\\_tools\\_oapi](http://www.mapquest.com/features/main.adp?page=developer_tools_oapi).

**Map24 AJAX** is a mapping service that allows a developer to place a Map24 map into an application with the use of JavaScript. The API documentation for programmatically adding these maps is available at <http://devnet.map24.com/index.php>.

**Microsoft's Virtual Earth** combines the features of its MapPoint web service with other GIS digital imagery to create a robust platform that can be used in the government or business sector. You can find the API documentation for using Virtual Earth at [http://dev.live.com/virtualearth/default.aspx?app=virtual\\_earth](http://dev.live.com/virtualearth/default.aspx?app=virtual_earth).

Yahoo! makes it easy to use its mapping API to create custom maps and embed them into an existing application. After obtaining an application ID from Yahoo!, using the Yahoo! Maps AJAX API library is a snap. The first thing to do is to include the library into your application, like so:

```
<script type="text/javascript"
    src="http://api.maps.yahoo.com/ajaxymap?v=3.0&appid=[Application Id]">
</script>
```

Once you have the library, you just create a placeholder element for the map that you can shape with CSS rules into whatever size you need. After that, simply decide whether to use latitude and longitude coordinates to specify the starting map location, or the built-in geocoding feature. Example 3.125 shows how to add a simple map control to an application that has some elementary tools included with it.

**Example 3.125. Adding a Yahoo! Map control using the Yahoo! Maps AJAX API library**

```
<html>
    <head>
        <script type="text/javascript"
            src="http://api.maps.yahoo.com/ajaxymap?v=3.0&appid=[Application Id]">
        </script>
        <style type="text/css">
            #mapContainer {
                height: 600px;
                width: 600px;
            }
        </style>
    </head>
    <body>
        <div id="mapContainer"></div>
        <script type="text/javascript">
            //<![CDATA[
            /* Create a latitude/longitude object */
            var myPoint = new YGeoPoint(38.64, -90.24);
            /* Create a map object */
            var map = new YMap(document.getElementById('mapContainer'));
            /* Add a map type control */
            map.addTypeControl();
            /* Set the map type to one of: YAHOO_MAP_SAT, YAHOO_MAP_HYB, or
             * YAHOO_MAP_REG */
            map.setMapType(YAHOO_MAP_REG);
            /* Add a pan control */
            map.addPanControl();
            /* Add a slider zoom control */
            map.addZoomLong();
            /* Display the map centered on a latitude and longitude */
            map.drawZoomAndCenter(myPoint, 3);
            //]]&gt;
        &lt;/script&gt;
    &lt;/body&gt;
&lt;/html&gt;</pre>
```

Its main features are the panning and zooming tools, and the ability to toggle among satellite, map, and hybrid modes.

A good addition to this type of map is the ability to add features that contain notations without having to know any extra programming. By having your script pass GeoRSS tagged files through the API interface, Yahoo! Maps will automatically add these features to the map. The World Wide Web Consortium (W3C) has a basic Geo (WGS84 lat/long) Vocabulary for defining these XML-based files, of which Yahoo! Maps takes advantage in RSS 2.0 format. You can find more information on this vocabulary at <http://esw.w3.org/topic/GeoInfo>. Yahoo! Maps

bases its XML file on the GeoRSS 2.0 standard, and uses channel and item elements to define the data. Example 3.126 shows an example of a GeoRSS feed.

#### **Example 3.126. A sample GeoRSS feed to use with Yahoo! Maps**

```
<?xml version="1.0" encoding="utf-8"?>
<rss version="2.0" xmlns:geo="http://www.w3.org/2003/01/geo/wgs84_pos#"
      xmlns:ymaps="http://api.maps.yahoo.com/Maps/V2/AnnotatedMaps.xsd">
  <channel>
    <title>Example RSS Data</title>
    <link><! [CDATA[http://www.oreilly.com]]></link>
    <description>Sample result</description>
    <ymaps:Groups>
      <Group>
        <Title>Museums</Title>
        <Id>museums</Id>
        <BaseIcon width="16" height="16">
          <! [CDATA[http://developer.yahoo.com/maps/star_blue.gif]]>
        </BaseIcon>
      </Group>
      <Group>
        <Title>Parks</Title>
        <Id>parks</Id>
        <BaseIcon width="16" height="16">
          <! [CDATA[http://developer.yahoo.com/maps/star_green.gif]]>
        </BaseIcon>
      </Group>
    </ymaps:Groups>
    <item>
      <title>St. Louis Art Museum</title>
      <link><! [CDATA[http://www.stlouis.art.museum/]]></link>
      <description>The St. Louis Art Museum</description>
      <ymaps:Address>1 Fine Arts Drive</ymaps:Address>
      <ymaps:CityState>St. Louis, MO</ymaps:CityState>
      <ymaps:GroupId>museums</ymaps:GroupId>
    </item>
    <item>
      <title>Missouri History Museum</title>
      <link>
        <! [CDATA[
          http://www.mohistory.org/content/HomePage/HomePage.aspx
        ]]>
      </link>
      <description>The Missouri History Museum</description>
      <ymaps:Address>5700 Lindell Blvd</ymaps:Address>
      <ymaps:CityState>St. Louis, MO</ymaps:CityState>
      <ymaps:GroupId>museums</ymaps:GroupId>
    </item>
    <item>
      <title>City Museum</title>
      <link><! [CDATA[http://www.citymuseum.org/home.asp]]></link>
      <description>City Museum</description>
      <ymaps:Address>701 N 15th St</ymaps:Address>
      <ymaps:CityState>St. Louis, MO</ymaps:CityState>
      <ymaps:GroupId>museums</ymaps:GroupId>
    </item>
    <item>
      <title>Forest Park</title>
      <link><! [CDATA[http://www.forestparkforever.org/HTML/]]></link>
      <description>Forest Park Forever</description>
      <ymaps:Address>5595 Grand Dr</ymaps:Address>
      <ymaps:CityState>St. Louis, MO</ymaps:CityState>
      <ymaps:GroupId>parks</ymaps:GroupId>
    </item>
    <item>
      <title>Tower Grove Park</title>
```

```

<link>
    <! [CDATA[http://stlouis.missouri.org/parks/tower-grove/] ]>
</link>
<description>Tower Grove Park</description>
<ymaps:Address>4256 Magnolia Ave</ymaps:Address>
<ymaps:CityState>St. Louis, MO</ymaps:CityState>
<ymaps:GroupId>parks</ymaps:GroupId>
</item>
</channel>
</rss>

```

To include this type of data in a map, simply add the following code to the JavaScript:

```

/* The sample overlay data from a GeoRSS file */
map.addOverlay(new YGeoRSS('http://www.holdener.com/maps/sample.xml'));

```

This is just a simple example of using the Yahoo! Maps API. Other map service APIs are similar in their ease of use, though they differ from each other in one way or another. Such a variety of mapping service APIs is available that it should not be too difficult to find one that meets your needs.

### 3.3.21.5 Music/Video Services

Many music and video capabilities are available on the Web today, from streaming video to Internet radio stations. In fact, so many exist that it is hard to even begin to list the choices available for developers in the form of web services. Both the business and private sectors have come to realize how effective music and video media on the Web can be. And not just for personal enjoyment, either, but also for sharing homemade or favorite sources of this media with the rest of the world.

Services such as YouTube were instant successes, and they have become a major source of community sharing for the world in the medium of streaming video. YouTube is so successful that a number of similar services made for a specific purpose are now thriving on the Web. Because of this popularity, it was only natural for a number of web services to be created that aid in the use and functionality of these sites.

**SeeqPod** is a web service that suggests music recommendations based on songs submitted to it. You can find the API documentation for access to the service at <http://www.seeqpod.com/api/>.

**Rhapsody** is a site that lets you stream music from its "browser" from within your web browser, and lets you search the music database that it keeps. The Rhapsody web service gives developers access to this technology on the site. The API documentation for using the service is located at <http://webservices.rhapsody.com/>.

**Last.fm** is the main site for the Audioscrobbler system, which collects data from people as they listen to music to track habits and song relationships, and makes those statistics available through its web service. You can find the API documentation for instructions on using this service at <http://www.audioscrobbler.net/data/webservices/>.

**YouTube** is a community portal that offers users the ability to view and share videos on the Web. YouTube offers an API to this service that can be put to use from within other web applications. The API documentation to utilize the service is at <http://www.youtube.com/dev>.

**Dave.TV** is a provider of video distributions in a community setting where users can share and view videos from the Web. Dave.TV offers a web API that allows for programmatic communication with its content delivery system. The API documentation for Dave.TV is at <http://dave.tv/Programming.aspx>.

A popular service on the Internet is the video sharing community of YouTube. Thanks to services such as this, users can easily search and view videos that are tagged by category from within other applications. There are no tricks to adding the functionality of YouTube and similar services, as they all provide fairly easy-to-use APIs for this purpose.

YouTube has both REST and XML-RPC access to its web service, though the following examples use REST. The service is easy to access and use, as it takes the following format to request data:

```
http://www.youtube.com/api2_rest?method=<method name>&dev_id=<developer id>
[&user=<YouTube user name>]
```

A successful response to this REST request takes the following format:

```
<ut_response status="ok">
    <!-- The XML for the response -->
<ut_response>
```

A request that ends in error, however, takes this form:

```
<ut_response status="fail">
    <error>
        <code>Code Number</code>
        <description>Description of code error.</description>
    </error>
</ut_response>
```

YouTube gives developers access to many requests—some dealing with user access and others dealing with video viewing. With the youtube.videos.list\_featured method as an example, a request to the service would look like this:

```
http://www.youtube.com/api2_rest?method=youtube.videos.list_featured&
dev_id=<developer id>
```

The response for this request looks like this:

```
<video_list>
    <video>
        <author>macpulenta</author>
        <id>y14g50q4hQ0</id>
        <title>Scarlett Johansson - Speed Painting</title>
        <length_seconds>412</length_seconds>
        <rating_avg>4.5</rating_avg>
        <rating_count>4476</rating_count>
        <description>
            A new speed painting in Photoshop. At this time, a beautiful woman...
            Enjoy it. And thanks for all your comments and messages to my other
            videos!! Gracias!!!
            (It was done with a digital tablet and the Background music is
            "Adagio for strings" by Dj Tiesto)
        </description>
        <view_count>859395</view_count>
        <upload_time>1121398533</upload_time>
        <comment_count>1883</comment_count>
        <tags>scarlett johansson speed painting photoshop</tags>
        <url>http://www.youtube.com/watch?v=y14g50q4hQ0</url>
        <thumbnail_url>
            http://static.youtube.com/get_still?video_id=y14g50q4hQ0
        </thumbnail_url>
        <embed_status>ok</embed_status>
    </video>
    ...
</video_list>
```

This response can be sent from a server-side script directly to a client script that requested it with an Ajax call. Then it is necessary to parse the information needed for the particular client page so that it can be displayed. Let's assume that the JavaScript code will be placing the data in an XHTML structure like this:

```
<div class="youTubeContainer">
    <div class="youTubeTitle"></div>
    <div class="youTubeThumb">
        <a href="">
```

```

        <img src="" alt="" title="" />
    </a>
</div>
<div class="youTubeDescription"></div>
</div>

```

Example 3.127 shows the JavaScript that handles the response from the server script and parses it to create the YouTube links in the application.

#### ***Example 3.127. Parsing a response from YouTube and putting the results into an XHTML application***

```

/* Parsing a response from YouTube and putting the results into an
 * XHTML application.
 * This function, getYouTubeResults, takes the /xhrResponse/'s responseXML and
 * parses it, placing the necessary elements into the output string that will be
 * the /innerHTML/ of /someElement/.
 * @param {Object} p_xhrResponse The XMLHttpRequest response from the server.
 * @return Returns whether the results were obtained correctly.
 * @type Boolean */
function getYouTubeResults(p_xhrResponse) {
    try {
        /* Get all of the video elements from the response */
        var videos = p_xhrResponse.responseXML.getElementsByTagName('video');
        var output = '';
        /* Loop through the video elements and format the output */
        for (var i = 0, il = videos.length; i < il; i++) {
            output += '<div class="youTubeContainer">';
            output += '<div class="youTubeTitle">' +
                videos[i].getElementsByTagName('title')[0].nodeValue + '</div>';
            output += '<div class="youTubeThumb"><a href="' +
                videos[i].getElementsByTagName('url')[0].nodeValue +
                '"><img src=' + videos[i].getElementsByTagName('thumbnail_url')[0].nodeValue +
                '" alt=' + videos[i].getElementsByTagName('title')[0].nodeValue +
                '" title=' + videos[i].getElementsByTagName('title')[0].nodeValue +
                '" /></a></div>';
            output += '<div class="youTubeDescription">' +
                videos[i].getElementsByTagName('description')[0].nodeValue + '</div>';
            output += '</div>';
        }
        /* Place the output in the document */
        $('someElement').innerHTML = output;
        /* Return true, indicating the function executed correctly */
        return (true);
    } catch (ex) {
        /* There was a problem retrieving video, let the user know and return false
         */
        $('someElement').innerHTML =
            'There was an error getting the YouTube videos.';
        return (false);
    }
}

```

### **3.3.21.6 News/Weather Services**

Being a news junkie, it is important to me to get as much news information as I can as quickly as I can. Unfortunately, there has never been one site that gives me everything I want to use. As the number and variety of news services and RSS feeds have increased, however, so has the ability to create my own news sites that aggregate everything I want to view together at one time.

Some services supply news directly from their sites and provide that data through their own services. And other services grab different RSS feeds and provide a single point at which a variety of news can be obtained.

The following are some examples of news services from 2008:

**NewsCloud** serves two purposes: it acts as a community for like-minded people to come together and express their ideas on corporate media and censorship, and it aggregates important news stories from around the Web. NewsCloud offers an API so that anyone can present his own NewsCloud data using his own applications. It is located at <http://www.newscloud.com/learn/apidocs/>.

**NewsIsFree** offers access to thousands of news sources through its portal, which allows for browsing news headlines. Currently featuring more than 25,000 news channels, NewsIsFree allows for programmatic access to creating, editing, and searching its sources through its API interface, which is located at <http://www.newsisfree.com/webservice.php>.

**NewsGator** is an information media company that deploys RSS aggregation solutions for all types of clients, from end users to corporate media companies. By providing an API to the resources NewsGator provides, anyone can develop applications to access and manipulate these resources outside of NewsGator. The API documentation is at <http://www.newsgator.com/ngs/api/overview.aspx>.

The **BBC** is a media broadcasting company in the United Kingdom that provides TV and radio feeds across the Internet. The BBC has provided an API to access its TV-Anytime database, which feeds TV and radio feeds. The documentation for this API is at <http://www0.rdtodo.bbc.co.uk/services/api/>.

**WeatherBug** displays the latest weather conditions for a specific area, and includes the current weather conditions, severe-weather alerts in the United States, daily forecasts, and much more. WeatherBug provides an API to these services so that this functionality can be used on custom applications. The WeatherBug API documentation is located at <http://api.weatherbug.com/api/>.

NewsIsFree for example uses a SOAP interface for all communication to its web service. To use this service, the developer must have a valid username and password, plus a personal application key.

It is simple to create a request to NewsIsFree using PHP and SOAP. First, create a client connection using PHP 5's built-in SOAP class, `SoapClient()`:

```
$client = new SoapClient('http://newsisfree.com/nifapi.wsdl',
    array('trace' => 1, 'exception' => 0));
```

Then, simply call the API method with the necessary parameters. For example, to call the API method `getNews()`, you would code the following:

```
$result = $client->getNews('[application key]',
    '[login]',
    '[password]',
    '[site id]',
    true);
```

For a `getNews()` request, the results are passed as an RSS feed, and they look like this:

```
<?xml version="1.0" encoding="UTF-8"?>
<rss version="0.92">
    <channel>
        <title>CNN</title>
        <link>/sources/info/2315/</link>
        <description>
            The world's news leader (powered by
            http://www.newsisfree.com/syndicate.php - FOR PERSONAL AND NON
            COMMERCIAL USE ONLY!)
        </description>
        <language>en</language><webMaster>contact@newsisfree.com</webMaster>
        <lastBuildDate>05/03/07 02:59 7200</lastBuildDate>
        <image>
            <link>http://www.newsisfree.com/sources/info/2315/</link>
            <url>http://www.newsisfree.com/HPE/Images/button.gif</url>
```

```

<width>88</width>
<height>31</height>
<title>Powered by NewsIsFree</title>
</image>
<item>
    <id>i,199624023,2315</id>
    <title>Armored truck robbers flee with $1.8 million</title>
    <link>
        http://rss.cnn.com/~r/rss/cnn_topstories/~3/113743933/index.html
    </link>
    <description>
        Read full story for latest details.&lt;p&gt;&lt;a href="http://rss.cnn.com/~a/rss/cnn_topstories?a=rX51Ch"&gt;
        &lt;img id="hpeitemad"
        src="http://rss.cnn.com/~a/rss/cnn_topstories?i=rX51Ch"
        border="0"&gt;&lt;/img&gt;&lt;/a&gt;&lt;/p&gt;
        &lt;img src="http://rss.cnn.com/~r/rss/cnn_topstories/~4/113743933"
        height="1" width="1"/&gt;
    </description>
    <read>0</read>
</item>
...
<item>
    <id>i,199609740,2315</id>
    <title>Dow on best streak since 1955</title>
    <link>
        http://rss.cnn.com/~r/rss/cnn_topstories/~3/113721652/index.htm
    </link>
    <description>
        The Dow Jones industrial average hit another record high Wednesday,
        capping its longest winning stretch in almost 52 years as investors
        welcomed strong earnings, lower oil prices, media merger news and a
        strong reading on manufacturing.&lt;p&gt;
        &lt;a href="http://rss.cnn.com/~a/rss/cnn_topstories?a=bodxMF"&gt;
        &lt;img id="hpeitemad"
        src="http://rss.cnn.com/~a/rss/cnn_topstories?i=bodxMF"
        border="0"&gt;&lt;/img&gt;&lt;/a&gt;&lt;/p&gt;&lt;img
        src="http://rss.cnn.com/~r/rss/cnn_topstories/~4/113721652"
        height="1" width="1"/&gt;
    </description>
    <read>0</read>
</item>
</channel>
</rss>

```

Example 3.128 shows how to construct a request to NewsIsFree for the latest CNN news. The results are passed to the client as XML to be parsed and displayed to the user.

#### **Example 3.128. Demonstrating how to use SOAP and PHP to pull data from the NewsIsFree web service**

```

<?php
/* Demonstrating how to use SOAP and PHP to pull data from the
 * NewsIsFree web service.
 * This example uses PHP 5's built-in SOAP class to request information from the
 * NewsIsFree web service. */
/* Set the parameters for the request query */
$app_key = '[application id]';
$login = '[login]';
$password = '[password]';
/* Site id 2315 is CNN */
$site = '2315';
$client = new SoapClient('http://newsisfree.com/nifapi.wsdl');
try {
    $result = $client->getNews($app_key, $login, $password, $site, true);
    /* Change the header to text/xml so that the client can use the return string

```

```

        * as XML */
header('Content-Type: text/xml');
/* Give the client the XML */
print($result[0]);
} catch (SoapFault $ex) {
    /* Something went wrong, show the exception */
    print($ex);
}
?>

```

### 3.3.21.7 Photo Services

When Flickr went live, it became an instant success and attracted millions of people who began to share their photos with the rest of the world. Since that time, other sites have been created with the same basic functionality, but with slightly different services. More important, communities exist within each of these sites, and they create many levels of communication and sharing between people all over the globe.

To increase their popularity and appeal to the development community, these photo services began to release APIs for their software so that outside web applications could reproduce the functionality offered on each site. These APIs help developers relatively easily do some pretty cool things in their web applications using web services like Flickr.

**Flickr** is a community photo sharing service that allows anyone to upload and share his photos with the rest of the world. The photos are tagged and categorized for easy searching and displaying of the Flickr data. A Flickr API and access to Flickr's public data is available; the documentation for the API is located at <http://www.flickr.com/services/>.

**SmugMug** is a fee-based photo sharing service that stores photos for its customers and allows access to the photos through its site. SmugMug offers an API for anyone to utilize its services and functionality. The documentation for this API is at <http://smugmug.jot.com/API>.

**Pixagogo** is a photo sharing and storage service that adds the ability to print the photos it stores. Pixagogo offers an API to its functionality so that you can build photo applications using its services. The documentation for Pixagogo's API is located at <http://www.pixagogo.com/Tools/api/apihelp.aspx>.

**Faces.com** is a community site that allows individuals to upload and share photos and music, blog posts, and more. Faces.com provides an API so that you can build applications utilizing its technology, and you can find it at <http://www.faces.com/Edit/API/GettingStarted.aspx>.

**Snipshot** is a service that allows online editing of images from its site, and the ability to save the images to a remote address. Through its API, Snipshot allows for the programmatic editing of images from custom applications. Documentation for this API is located at <http://snipshot.com/services/>.

Flickr is one of the more popular services. It allows developers to request information from its web service in the REST, XML-RPC, and SOAP formats. An example REST request would look like this:

```
http://api.flickr.com/services/rest/?method=flickr.test.echo&name=value
```

Generally, the response format will be the same as that of the request, so a REST request would get a REST response. You can change the default response type using the `format` parameter in the request:

```

<?php
/* Set up the parameters for the request */
$params = array(
    'api_key'      => '[API key]',
    'method'       => 'flickr.blogs.getList',
    'format'       => 'json'
);
$encoded_params = array( );
/* Loop through the parameters and make them safe */

```

```

foreach ($params as $param => $value)
    $encoded_params[] = urlencode($param) . '=' . urlencode($value);
?>

```

There will be some debate as to which of Flickr's response formats is the easiest to use. The choices are formatted XML with REST, XML-RPC, and SOAP; JavaScript Object Notation (JSON); and serialized PHP. The REST response is an easy response format to use, as it is a simple XML block. JSON, however, also has its advantages in size and structure. A successful JSON response is in this format:

```

jsonFlickrApi({
    "stat": "ok",
    "blogs": {
        "blog": [
            {
                "id" : "23",
                "name" : "Test Blog One",
                "needspassword" : "0",
                "url" : "http://blogs.testblogone.com/"
            },
            {
                "id" : "76",
                "name" : "Second Test",
                "needspassword" : "1",
                "url" : "http://flickr.secondtest.com/"
            }
        ]
    }
});
```

Meanwhile, if an error occurs, the JSON returned looks like this:

```

jsonFlickrApi({
    "stat" : "fail",
    "code" : "97",
    "message" : "Missing signature"
});
```

Many methods are available with the Flickr API. These methods can be separated by functionality, which you can see by breaking down the Flickr method names. All Flickr methods begin with the `flickr` namespace, followed by the function category the method is in—activity, auth, favorites, and so on. The Flickr API covers just about anything a developer would want to do programmatically. Example 3.129 demonstrates how to make a REST request to the Flickr web service and get a JSON response that is sent to the client. The example gets the title of a specific photo on Flickr's site that can be returned to the client.

#### **Example 3.129. Making a REST call to the Flickr web service and getting a PHP response**

```

<?php
/** Making a REST call to the Flickr web service and getting a PHP response.
 * This file demonstrates how to send a request to Flickr's API methods and
 * send the response to the client using the REST architecture and JSON. */
/* Set up the parameters for the request */
$params = array(
    'api_key' => '[API key]',
    'method' => 'flickr.blogs.getList',
    'format' => 'json'
);
$encoded_params = array( );
/* Loop through the parameters and make them safe */
foreach ($params as $param => $value)
    $encoded_params[] = urlencode($param) . '=' . urlencode($value);
/* Make the API request */
$url = "http://api.flickr.com/services/rest/?".implode('&', $encoded_params);
$response = file_get_contents($url);

```

```

/* Send the response to the client to parse */
print($response);
?>

```

Example 3.130 demonstrates how easy it is to make the JSON response usable on the web page. It is also easy to modify this function to do something different from what it is doing.

#### **Example 3.130. Demonstrating how to handle a JSON response from Flickr**

```

/* Demonstrating how to handle a JSON response from Flickr.
 * This function, jsonFlickrResponse, takes the JSON response from the server
 * and parses the results by creating <div> elements to place the blog name in.
 * It sends the error code and message should an error occur in the request.
 * @param {Object} p_xhrResponse The XMLHttpRequest response from the server. */
function jsonFlickrResponse(p_xhrResponse) {
    var rsp = p_xhrResponse.responseText;
    /* Did we get a valid response? */
    if (rsp.stat == 'ok') {
        /* Loop through the log records */
        for (var i = 0, il = rsp.blogs.blog.length; i < il; i++) {
            var blog = rsp.blogs.blog[i];
            var div = document.createElement('div');
            var txt = document.createTextNode(blog.name);
            div.appendChild(txt);
            document.body.appendChild(div);
        }
        /* Did we get a fail message? */
    } else if (rsp.stat == 'fail') {
        var div = document.createElement('div');
        var txt = document.createTextNode(rsp.code + ': ' + rsp.message);
        div.appendChild(txt);
        document.body.appendChild(div);
    }
}

```

### **3.3.21.8 Reference Services**

The Web is one great library, used to look up everything from antidisestablishmentarians to zooarchaeologists. OK, maybe not everything, but a great deal of content on the Web can be used as reference or for lookup. When sites such as Wikipedia arrived on the scene, the volume of information that could be referenced in one place skyrocketed.

Reference content is more than just articles on Wikipedia, however, a site which closely resembles encyclopedias of the past. The amount of information that can be gathered, from demographics to genealogical data, is really the reference information that I mean. Unfortunately, some of this information is still difficult to get in a single, usable form. From this chapter's point of view, many reference sites still lack access as web services—even sites such as Wikipedia (at least as of the time of this writing, though a web service is in the works).

**RealEDA Reverse Phone Lookup** provides an interface to the names and addresses that are associated with any telephone number. This fee-based API allows outside applications to access and incorporate this information. The resources for this API are at <http://www.strikeiron.com/productdetail.aspx?p=157>.

**ISBNdb** service is a database containing book information taken from libraries around the world and provides research tools to this data. ISBNdb provides a developer API that allows for data requests from remote applications; its documentation is located at <http://isbndb.com/docs/api/index.html>.

**Urban Dictionary** is a dictionary that provides definitions for modern-day slang. It provides an API so that you can look up words from within custom applications. You can find the documentation to this API at <http://www.urbandictionary.com/tools.php>.

**SRC Demographics** is a service providing access to demographic information based on the 2000 U.S. census. The API that is provided allows for applications deployed on any web site to seamlessly integrate this information. The documentation for this API is at <http://belay.extendtheresearch.com/api/help/>.

**StrikeIron U.S. Census** is a fee-based service that allows for the retrieval of extensive information from the 2000 U.S. census. The API to the service allows for this information to be placed in custom web applications. The API documentation is located at <http://www.strikeiron.com/developers/default.aspx>.

**StrikeIron Residential Lookup** is a fee-based service allowing for the retrieval of residential information within the U.S. and Puerto Rico. The API allows for programmatic interfaces to this information that can be added to a web application. The API documentation is located at <http://www.strikeiron.com/developers/default.aspx>.

Something that is beginning to change, in small steps anyway, is the online availability of public information from local, state, and federal government agencies. However, a great deal of information still is not available on the Internet, or if it is, fees are associated with it. Only time will tell how much information will actually become available on the Internet, and what it will cost to access it.

Then there is the problem of sites that have good reference information available, whether or not a fee is involved. However, these sites have not created web services to access their information from outside their site frameworks. A major area where this is lacking is genealogical information. Again, in time, accessing reference information through web services will improve. We'll just have to wait to see how long this will take.

### 3.3.21.9 Search Services

I think it is pretty safe to say that if you have been on the Web, you have conducted a search of some kind. Let's face it, it can be pretty hard to find the content you are looking for, especially as the Web gets older and more mature. There is so much information on the Web, even within a single site, that you would be hard-pressed to find what you were looking for without performing at least one search.

This was not always the case. In the late 1990s, it was much easier to find specific content simply because the Web did not have the volume of information it has today (this assumes that the content you were looking for even existed!). The reverse is true today; the content is out there, somewhere, but it is much harder to find.

The Web relies on searching, whether it's from a site such as Google or Yahoo! or from within a site; blogs are an example of sites that need internal searching. Having APIs to some of the better search engines available can greatly enhance a web application; developers can add the searching directly into the application, saving trips to other places.

**Google AJAX Search API.** Google is the most popular search engine on the Web, and it has been a leader in the web service API arena from the start. By providing an API to its searching capabilities, you can easily add Google Search to any web application. The documentation for this API is available at <http://code.google.com/apis/ajaxsearch/>.

**Yahoo! Search** is another popular search engine on the Web that offers an API to access its search capabilities from an outside application. The documentation for the API is located at <http://developer.yahoo.com/search/web/>.

**Windows Live Search** is part of Microsoft's Live suite of services and allows developers deeper control of Windows Live search functionality and social relationships through its API. The documentation for this API is located within MSDN at <http://msdn2.microsoft.com/en-us/library/bb264574.aspx>.

I touched on search service APIs in [Chapter 16](#) where I explored some of the capabilities of Google's AJAX Search API. The better APIs available are just as easy to use as Google's is, and should not cause much trouble for developers to integrate them into an application.

### 3.3.21.10 Shopping Services

Most business owners have recognized by now that if you have something to sell, you better have a way to sell it online. This is, in no small part, thanks to sites such as Amazon and eBay, which have made shopping on the Web simple, quick, and painless.

Instead of investing money into their own sites and shopping technology, many small to medium-size businesses have turned to larger commercial sites for hosting and the technology to drive their business. One disadvantage to this is a slight loss of brand recognition, as consumers will see what is driving shopping carts with less emphasis on the business that brought them there. The ability to use these sites from a separate application or web site through APIs has changed all that.

Web services that enable developers to access the functionality of these larger e-commerce sites allow for better site integration and smoother, more seamless business applications. The following is a list of some of the shopping applications available:

**Amazon** has a suite of web services available to developers. The Amazon E-Commerce Service provides product data and e-commerce functionality through an API that developers can utilize within an application. This API documentation is located at

[http://www.amazon.com/gp/browse.html/ref=sc\\_fe\\_1\\_2/103-1644811-9832630?%5Fencoding=UTF8&node=12738641&no=3435361](http://www.amazon.com/gp/browse.html/ref=sc_fe_1_2/103-1644811-9832630?%5Fencoding=UTF8&node=12738641&no=3435361).

**DataUnison eBay Research** service provides marketing data on eBay customer buying and selling trends. Its API gives developers access to eBay's main categories and subcategories to pull all relevant information into separate web applications. You can find the documentation for this API at <http://www.strikeiron.com/ProductDetail.aspx?p=232>.

**UPC Database** provides a database of contributor-provided UCC-12 (formerly UPC) and EAN/UCC-13 codes. With the provided API, developers can look up codes to products from within existing applications. The API documentation is at <http://www.upcdatabase.com/xmlrpc.asp>.

**eBay** provides a marketplace for more than 200 million users that buy and sell on the site. With the accessibility of eBay's API, developers can access all of eBay's functionality from outside applications with ease. The starting place for developers interested in different languages that work with eBay's API is at <http://developer.ebay.com/>.

**CNET** is a data center for electronic products such as digital cameras, computers, and MP3 players, as well as a collection of software titles. CNET provides an API interface to functionality that interfaces with its services and that developers can integrate into other applications. The documentation to this API is located at <http://api.cnet.com/>.

As a quick example of using a shopping service, let's look at eBay. eBay has done an awesome job of documenting its API and providing examples on how to use it properly. The API can be called using REST or SOAP, and eBay provides examples on how to use the API with Java, PHP, C#, and ASP, among others.

The following is an example of a REST request to eBay:

```
http://rest.api.ebay.com/restapi?CallName=GetSearchResults&
RequestToken=[UserToken]&RequestUserId=[UserName]&Query=[Query Words]&
Version=491&UnifiedInput=1
```

This example calls the method `GetSearchResults`, and expects a maximum of five results. An eBay `UserName` and `UserToken` are required to use this service.

Every eBay REST API method requires the same basic parameters when a client makes a request. The best way to understand how eBay's web service works is to see how the server side would make a request to the server when a variable is passed to it. Example 3.131 shows how to handle this in PHP.

### **Example 3.131. Requesting search results from the eBay REST API**

```
<?php
/** Requesting search results from the eBay REST API.
 * This file is an example of searching eBay using its REST API, utilizing the
 * CURL package to make the request and return the results. */
/* Was a search request received? */
if (isset($_REQUEST['search_text'])) {
    $request_token = '[UserToken]';
    $request_user_id = '[UserName]';
    $query = $_REQUEST['search_text'];
    /* Create the REST string */
    $rest_request = 'http://rest.api.ebay.com/restapi?'
        .'RequestToken='.$request_token
        .'&RequestUserId='.$request_user_id
        .'&CallName=GetSearchResults'
        .'&Schema=1'
        .'&Query='.urlencode($query).'\''
        .'&MaxResults=10'
        .'&SearchInDescription=1';
    try {
        $curl_request = curl_init();
        /* Set the URL to post the request to */
        curl_setopt($curl_request, CURLOPT_URL, $rest_request);
        /* This allows for errors to be handled */
        curl_setopt($curl_request, CURLOPT_FAILONERROR, 1);
        /* This allows for redirection */
        curl_setopt($curl_request, CURLOPT_FOLLOWLOCATION, 1);
        /* This sets the response to be set into a variable */
        curl_setopt($curl_request, CURLOPT_RETURNTRANSFER, 1);
        /* This sets a timeout for 30 seconds */
        curl_setopt($curl_request, CURLOPT_TIMEOUT, 30);
        /* This sets the post option */
        curl_setopt($curl_request, CURLOPT_POST, 0);
        /* Execute the CURL process, and set the results to a variable */
        $result = curl_exec($curl_request);
        /* Close the connection */
        curl_close($curl_request);
    } catch (Exception $ex) {
        $result = $ex.message;
    }
} else
    $result = "A search request was not sent.";
print ($result);
?>
```

Example AJA.E19 uses the CURL package, which must be installed on the web server on which the code will be executed. You will notice that I set some additional parameters as part of the GetSearchResults request. Here is a list of the parameters available to set with this method: Query, Category, ItemTypeFilter, PayPal, SearchInDescription, LowestPrice, HighestPrice, PostalCode, MaxDistance, Order, MaxResults, and Skip. Any of these could be sent as part of the request from the client; the server-side script would simply need to be modified to handle those `$_REQUEST` parameters as well.

#### **3.3.21.11 Other Services**

Of course, other services are available, but they do not fit well into a broad category. Furthermore, they range in functionality and complexity. If a developer needs a service, more than likely she will be able to find it on the Web somewhere with a bit of investigating. If a service has not been created yet, a few posts to forums might reveal someone else with the same needs. Collaborative efforts often produce great results.

**English Standard Version (ESV) Bible Lookup** provides a way for users to search and read the ESV Bible online. The ESV Bible Lookup provides an API to access its functionality from outside sources, the documentation for which you can find at <http://www.gnpbc.org/esv/share/services/api/>.

**Amnesty International** has provided a service that allows users to search for documents written about the freedom of expression, especially on the Internet. An API is provided to developers so that they can build custom applications that use the same database of content used by Amnesty International's campaign. The documentation for this API is located at <http://irrepressible.info/api>.

**411Sync** enables developers to have keyword searches available through mobile technology, giving sites better exposure to end users. The API for this functionality is available at <http://www.411sync.com/cgi-bin/developer.cgi>.

**Windows Live Custom Domains** service allows developers to programmatically manage their Windows Live Custom Domains user base. The API documentation for this functionality is available at <http://msdn2.microsoft.com/en-us/library/bb259721.aspx>.

**Sunlight Labs** provides clerical information about members of the 110th U.S. Congress, such as phone number, email address, district, and so on. An API facilitates this functionality; you can find the documentation at <http://sunlightlabs.com/api/>.

**Food Candy** is a social networking system for people who live for food. An API is available that helps developers with the functionality required of any social networking application. The API documentation is located at <http://www.foodcandy.com/docs/html/index.html>.

**Facebook** is a social networking system that allows friends to keep in contact with one another online. Facebook provides an API that allows developers to programmatically add content to a Facebook account from outside applications. The documentation for this API is at <http://developers.facebook.com/>.

### **3.3.23 Amazon Web Services**

Building on the success of its storefront and fulfillment services, Amazon now allows businesses to "rent" computing power, data storage and bandwidth on its vast network platform. This section demonstrates how developers working with small- to mid-sized companies can take advantage of Amazon Web Services (AWS) such as the Simple Storage Service (S3), Elastic Compute Cloud (EC2), Simple Queue Service (SQS), Flexible Payments Service (FPS), and SimpleDB to build web-scale business applications. With AWS, Amazon offers a new paradigm for IT infrastructure: use what you need, as you need it, and pay as you go.

#### **3.3.23.1 Infrastructure in the Cloud**

The Web has also become an open platform on which powerful services and applications can be built by established companies and newcomers alike. It is a very accessible platform that allows even small companies to create web applications and build a business without requiring the backing of a large enterprise. A person or group with some expertise, some time, and a good enough idea can create a web application that competes with the offerings of larger corporations—or even carves out an entirely new market. On the Web, the size and marketing clout of a large corporation does not guarantee it a monopoly on the attention and patronage of a global audience.

The Web is full of opportunities for companies both large and small, but the smaller companies face a difficult problem: infrastructure. Web applications that are popular and have thousands of users require significant infrastructure to provide the high performance and smooth experience that users demand. Industrial-strength infrastructure is very expensive to buy and maintain, so smaller companies with fewer users are often forced to do without. The difference between a web application serving a few dozen users and serving thousands may be no more than a glowing article and a few hours' time.

Although this kind of attention may be exactly what you hope for, unless you have invested heavily in infrastructure, your application may not survive the onslaught. On the other hand, if you spend too much money on servers, bandwidth, hosting, and the management of all this infrastructure, there will be little left to develop the application itself. A dilemma facing many small development teams is how to strike the right balance between investing in application development and funding robust and scalable infrastructure.

Amazon offers a new and compelling solution to this dilemma in the form of infrastructure web services. These services allow application developers to avoid altogether the burden of buying and maintaining physical infrastructure by making it possible to rent virtual infrastructure instead. In this book we will show you how you can build your applications on top of Amazon's services and effectively outsource your infrastructure.

**Amazon Simple Storage Service (S3)** offers secure online storage space for any kind of data, providing an alternative to building, maintaining, and backing-up your own storage systems. It makes your data accessible to any other applications or individuals you allow from anywhere on the Web. There are no limits on how much data you can store in the service, how long you can store it, or on how much bandwidth you can use to transfer or publish it. S3 is a scalable, distributed system that stores your information reliably across multiple Amazon data centers, and it is able to serve it quickly to massive audiences. Its storage application programming interface (API) is deliberately simple and makes no assumptions about the nature of the data you are storing. This simplicity means you can maintain complete control over how your data is represented in the service.

**Amazon Elastic Compute Cloud (EC2)** makes it possible to run multiple virtual Linux servers on demand, providing as many computers as you need to process your data or run your web application without having to purchase or rent physical machines. In EC2 you have full control over each server with root access to the operating system (the root user is the ultimate system administrator on Linux machines), a configurable firewall to manage network access, and the freedom to install any software you please. Once you have set up an EC2 server the way you like it, you can save it permanently as a server image. You can then launch new servers from this image to create virtual machines that are preconfigured and ready to do your bidding. The EC2 service offers computing resources that are very flexible. You can run as many servers as you need for as long as you need them, and you can shut them all down when they have served their purpose. The service offers an API to start and stop server instances, apply access and networking permissions, and manage your server images. You manage each individual server using standard Linux tools over a secure shell session.

**Amazon Simple Queue Service (SQS)** delivers short messages between any computers or systems with access to the Internet, allowing the components of your distributed web applications to communicate reliably without you having to build or maintain your own messaging system. With SQS you can send an unlimited number of messages via an unlimited number of message queues, and you can configure the performance characteristics and access permissions for each queue. The service uses a message locking and timeout mechanism that helps prevent

messages from being delivered more than once, while still ensuring they will be delivered despite any component failures or network dropouts. SQS is implemented as a distributed application within Amazon. Your messages are stored redundantly across multiple servers and data centers. The service's API allows you to send and receive messages, and to control their full life cycle.

**Amazon Flexible Payments Service (FPS)** transfers money between individuals or companies that have Amazon Payments accounts, allowing you to build applications that provide an online store or that implement a marketplace between customers and third-party vendors. With FPS you can make payments from traditional sources, such as credit cards and bank accounts, or from sources internal to Amazon Payments accounts that have lower fees and are designed to make micro-payment transactions feasible. All transactions need to be authorized by everyone involved in the transaction. The parties involved can impose detailed constraints on transactions, such as how and when transactions can be performed, how much money can be transferred, and who can send and receive the funds. Customers interact with your FPS application through an Amazon Payments gateway using their Amazon.com account. Because the transactions are mediated by Amazon, your customers are not required to provide you with their personal banking information, and you do not have the burden of securely storing this highly sensitive information.

**Amazon SimpleDB (SimpleDB)** stores small pieces of textual information in a simple database structure that is easy to manage, modify and search. If your application relies on a relatively simple database, this service can replace your traditional relational database (RDBMS) server leaving you with one less piece of infrastructure to purchase and maintain. SimpleDB is designed to minimize the complexity and administrative overhead involved in managing your data. It does not require a pre-defined schema so you can alter the structure and content of your database whenever you need to. It indexes every piece of information you store so all your queries run quickly. And it stores your data securely, redundantly and safely within Amazon's network of data centers.

These five web services—S3, EC2, SQS, FPS, and SimpleDB—share the same fundamental characteristics. They are pay-as-you-go, meaning you pay predictable fees based on how much or how little you use the service. There are no initial costs to join, no long-term subscription payments, and the usage fees are attractively low. The services are highly scalable, performing equally well in modest or massively demanding usage scenarios. This means that the applications built on them can be similarly scalable and are able to grow rapidly at short notice without hitting limits imposed by insufficient infrastructure. One significant feature is that all the services are designed to be highly reliable and fault-tolerant: the services and data resources are distributed across multiple servers and data centers within Amazon's infrastructure, and they are managed by a company with significant experience and investments in the operation of a global web business. To use AWS you first need to register for an account and provide a credit card to be billed for your service usage.

Here are the home pages for the infrastructure services we discuss in this book:

- Simple Storage Service (S3) <http://www.aws.amazon.com/s3>
- Elastic Compute Cloud (EC2) <http://www.aws.amazon.com/ec2>
- Simple Queue Service (SQS) <http://www.aws.amazon.com/sqs>
- Flexible Payments Service (FPS) <http://www.aws.amazon.com/fps>
- SimpleDB <http://www.aws.amazon.com/sdb>

Initially, the AWS infrastructure services were not conceived as products to be sold to developers external to Amazon but were instead designed to meet specific needs within Amazon's own internal systems. It was only later that these services were opened up to the public. The key implementation details of the services are therefore intended primarily to serve Amazon's needs and not necessarily use the methodologies or techniques common in the rest of the industry. Appreciating the reasoning behind the architectural decisions and their implementation details can help you to adjust your expectations for the services. This, in turn, will make it easier to design applications that work well with the services' capabilities.

Amazon's services are designed to power the Amazon.com web site and related partner applications. The services operate as small component cogs in a large service-oriented architecture (SOA). Each service performs a specific task as simply and efficiently as possible, while the strengths of many different services are combined as required to perform complex processes and build the rich Amazon.com web pages with which we are familiar. Amazon's SOA has been developed over many years of hard-won experience to be highly scalable to meet growing demand and be highly reliable despite the inevitable hardware and network failures that will occur in such an environment.

Amazon does not provide Service Level Agreements (SLAs) for all its services. This means that it does not always define the level of service it will guarantee to AWS customers, nor does it offer compensation should the level of service fall below expectations. Of the five infrastructure services discussed in this book, only S3 has an

SLA. For some organizations, the lack of a formal service agreement will make the AWS offerings too risky to accept. Even for groups or individuals without such strict requirements, the lack of an SLA can be disquieting, because Amazon generally only promises "best-effort" service. In addition to this, the AWS Terms and Conditions agreement permits the termination of AWS accounts at the sole discretion of Amazon. In legal terms, there does not seem to be a strong commitment by Amazon to providing high-quality service to AWS customers.

Developers of web applications that store, transmit, or process sensitive information need to be mindful of their responsibility to protect the security of this information and keep it private. Exposing this information to a third party, like Amazon, by using AWS infrastructure services can potentially reduce the degree of control you can maintain over the data, and transmitting the information to and from these services over the Internet can also pose a risk. In the effort to maintain the security and privacy of your data, there are three things you need to consider:

- User authentication;
- Secure transmission; and
- Secure storage

AWS infrastructure services are made available through three separate APIs: REST, Query, and SOAP.

**REST interfaces** offered by AWS use only the standard components of HTTP request messages to represent the API action that is being performed. These components include:

- HTTP method: describes the action the request will perform
- Universal Resource Identifier (URI): path and query elements that indicate the resource on which the action will be performed
- Request Headers: pieces of metadata that provide more information about the request itself or the requester
- Request Body: the data on which the service will perform an action

Web services that use these components to describe operations are often termed *RESTful* services, a categorization for services that use the HTTP protocol as it was originally intended.

**Query interfaces** offered by AWS also use the standard components of the HTTP protocol to represent API actions; however these interfaces use them in a different way. Query requests rely on parameters, simple name and value pairs, to express both the action the service will perform and the data the action will be performed on. When you are using a Query interface, the HTTP envelope serves merely as a way of delivering these parameters to the service. To perform an operation with a Query interface, you can express the parameters in the URI of a GET request, or in the body of a POST request. The method component of the HTTP request merely indicates where in the message the parameters are expressed, while the URI may or may not indicate a resource to act upon. These characteristics mean that the Query interfaces cannot be considered properly RESTful because they do not use the HTTP message components to fully describe API operations. Instead, the Query interfaces can be considered REST-like, because although they do things differently, they still only use standard HTTP message components to perform operations.

**SOAP interfaces** offered by AWS use XML documents to express the action that will be performed and the data that will be acted upon. These SOAP XML documents are constructed as another layer on top of the underlying HTTP request, such that all the information about the operation is moved out of the HTTP message and encapsulated in the SOAP message instead. For operations performed with a SOAP interface, the HTTP components of the request message are nearly irrelevant: all that is important is the XML document sent to the service as the body of the request. The valid structure and content of SOAP messages are defined in a Web Service Description Language (WSDL) document that describes the operations the service can perform, and the structure of the input and output data documents the service understands. To create a client program for a SOAP interface, you will typically use a third-party tool to interpret the WSDL document and generate the client stub code necessary to interact with the service. The approach used in the SOAP interfaces are very different from those used by the REST and Query interfaces. Operations expressed in SOAP messages are completely divorced from the underlying HTTP message used to transmit the request, and the HTTP message components, such as method and URI, reveal nothing about the operation being performed.

Each interface provides a slightly different way of interacting with a service, though the underlying API operations are largely the same. Table 3.28 shows the interfaces exposed by each service.

**Table 3.28. APIs made available by AWS services**

Service	REST API	Query API	SOAP API
S3	Yes	No	Yes
EC2	No	Yes	Yes
SQS	Yes	Yes	Yes
FPS	No	Yes	Yes
SimpleDB	No	Yes	Yes

AWS uses **XML documents** to represent structured information that is sent to, or received from, a service. Although the REST-based service interfaces rely less on XML documents than the corresponding SOAP interface, it is still vital to create and interpret XML documents to interact with the services.

Here is an XML document returned by the S3 service when we ask it for a listing of our data storage buckets:

```
<?xml version='1.0' encoding='UTF-8'?>
<ListAllMyBucketsResult xmlns='http://s3.amazonaws.com/doc/2006-03-01/'>
  <Owner>
    <ID>1a2b3c4d5e6f1a2b3c4d5e6f1a2b3c4d5e6f1a2b3c4d5e6f1a2b3c4d5e6f1a2b</ID>
    <DisplayName>jamesmurty</DisplayName>
  </Owner>
  <Buckets>
    <Bucket>
      <Name>oreilly-aws</Name>
      <CreationDate>2007-09-14T08:20:49.000Z</CreationDate>
    </Bucket>
    <Bucket>
      <Name>my-bucket</Name>
      <CreationDate>2007-09-24T08:39:30.000Z</CreationDate>
    </Bucket>
  </Buckets>
</ListAllMyBucketsResult>
```

The first line in this document tells us that it is indeed XML, that it complies with the version 1.0 XML specification, and that it is encoded using the UTF-8 unicode text encoding standard. All AWS XML documents use UTF-8 encoding. After the first line, we see the root element of the document (`ListAllMyBucketsResult`), which contains two further elements: `Owner` and `Buckets`. Each of these elements contains more subelements. The XML documents returned by AWS contain a hierarchical structure of information that is accessible using XPath queries.

### 3.3.23.2 S3: Simple Storage Service

S3 provides unlimited online storage space for files or data of any kind. Information stored in S3 is accessible from anywhere you have an Internet connection and is maintained in a highly scalable and reliable system. You can use S3 to securely store your personal data, to cheaply distribute content to the general public, or as a data storage component in a distributed web application architecture. Amazon offers a Service Level Agreement for S3 that makes users eligible for service credits should the S3 uptime percentage fall below 99.9%. To claim these credits, users of the service must track any faults experienced by their applications due to S3 downtime, and they must provide Amazon with detailed logging documentation to corroborate the claim.

S3's data model is very simple, comprising only two kinds of storage resource: objects and buckets. Objects store data and metadata, and buckets are containers that can hold an unlimited number of objects.

In addition to data storage, S3 provides access control mechanisms that allow you to keep your information private or make it public and accessible to anyone on the Internet. Access control settings are configured using a list of rules that describe who will be granted access to a resource and the kinds of access that will be permitted. Access control settings can be applied to both bucket and object resources.

Resources in S3 are identified using standard Universal Resource Identifiers (URIs). This means that publicly accessible objects can be downloaded from a URI resembling any standard web site location, such as

<http://s3.amazonaws.com/bucket-name/object-name>. S3 also allows resources to be accessed using alternative domain names. This feature allows you to publish links to your resources based on your own domain name, such as <http://www.mysite.com/object-name>, instead of the default S3-service domain name.

The data is stored redundantly within this architecture, spread across multiple physical servers and across multiple data centers in different locations. Amazon provides S3 data-center locations in the United States and Europe. This storage strategy provides huge benefits in terms of redundancy, reliability, and scalability, but it also leads to some drawbacks that you must consider when building applications that use S3:

- S3 Objects cannot be manipulated like standard files
- Changes take time to propagate
- S3 requests will fail occasionally
- S3's IP addresses may change over time

S3 account holders are billed monthly for their usage of the service. The service charges are based on three aspects of your S3 usage: the storage space consumed, the volume of data transferred to or from S3, and the number of API request operations that have been performed on your account. The fees can vary depending on the geographical location where you have chosen to store your data.

The **REST API interface** for the S3 service uses five HTTP methods to perform API operations: GET, HEAD, PUT, DELETE, and POST. The meaning of each method varies slightly, depending on what kind of S3 resource the operation is targeting: an object, a bucket, an Access Control List (ACL), or the S3 service itself. Table 3.29 lists some of the operations you can perform on S3 resources using different HTTP methods.

**Table 3.29 Acting on S3 resources with HTTP methods**

Resource	GET	HEAD	PUT	DELETE	POST
S3 Service	List your buckets	-	-	-	-
Bucket	List the bucket's objects	-	Create the bucket	Delete the bucket	-
Object	Retrieve the object's data and metadata	Retrieve the object's metadata	Create or replace the object	Delete the object	Create or replace the object
ACL (for a Bucket or Object resource)	Retrieve ACL settings	-	Apply new ACL settings	-	-

Resources in the S3 service are identified using URIs that include three components:

- The location of the S3 service and the communication protocol (HTTP or HTTPS)
- The S3 resource the request is targeting, such as a bucket or an object
- Optional request parameters used to identify special resources, such as the access control settings associated with a resource

To perform an action on an S3 resource, you must first construct a URI string that identifies that resource. Constructing this URI is not a trivial task, because its content can vary a great deal depending on what kind of resource is involved and whether the request will use the standard S3 service domain or an alternative hostname instead.

S3 understands three different URI formats:

1. A URI with the default service hostname `s3.amazonaws.com`, which will contain the bucket name in its resource path, if a bucket or object is specified.
2. A URI with a subdomain hostname constructed from the bucket name and `s3.amazonaws.com`; a bucket name will not be included in the resource path.
3. A URI with a virtual hostname that matches the name of the bucket; a bucket name will not be included in the resource path.

An **S3 bucket** is a container for data objects. A bucket does not contain any data; it is little more than a convenient way of grouping objects together. The closest computer system analogy to an S3 bucket would be a disk drive.

The access control permissions for each bucket can be configured to determine who can view the bucket's contents, or add and remove objects in the bucket. Buckets are also used as the basis for the simple access-logging capabilities provided by S3.

You can configure your buckets to be based in different geographical locations. At the time this book was written, Amazon provided S3 data centers in two locations: the United States and Europe. When a bucket is based in a location, all the objects created inside that bucket are automatically stored in that location. Making your S3 resources available from a specific location can improve the performance of S3 for customers living in that region. To work most efficiently over multiple locations, S3 uses alternative DNS names and request redirection techniques to ensure that service requests are sent to data centers in the region where the bucket is stored.

The use of alternative DNS names means that S3 clients must use the subdomain host-naming format in requests that refer to buckets located outside the United States, and that the names of these buckets must be compatible with the DNS system. If an S3 client refers to a non-U.S. bucket in the path of a request URI, instead of in a subdomain, the service will respond with a Permanent Redirect message (HTTP status 301). This response indicates that the client has used an inappropriate resource reference and must correct the reference before submitting a new request. The best way to avoid sending inappropriate references is to use the subdomain host-naming format whenever possible.

When a bucket is first created, or when it is deleted and recreated in a different location, there may be a delay while the bucket's location information is propagated through DNS. During this time, requests may be routed to the wrong location and S3 will have to send Temporary Redirect (HTTP code 307) responses to inform the client of the correct location. When an S3 client receives a Temporary Redirect response, it should resend the request to the location given in the redirect message.

A bucket is created by sending a PUT request to a URI specifying the name of the bucket you wish to create; the bucket's name is contained in the URI's path or as a subdomain of the request's target hostname. When the request is successful, the S3 service responds with an empty HTTP 200 response message. A bucket can be re-created if it already exists, in which case the bucket's creation date is updated to the current time, and its access control permissions are reset to the private default values. If you wish to create a bucket in a region other than the US location, you must include an XML configuration document in the body of the PUT request specifying the desired location. If you wish to create the bucket in the default US location, the PUT request must have an empty body.

Here is the XML configuration document that is included with the PUT request to specify a bucket's location. The bucket's location is identified by a code value in the LocationConstraint element. As of early 2008 the only location available outside the United States was Europe, which is represented by the location constraint code EU.

```
<CreateBucketConfiguration xmlns='http://s3.amazonaws.com/doc/2006-03-01/'>
  <LocationConstraint>EU</LocationConstraint>
</CreateBucketConfiguration>
```

To find out where one of your buckets is located, you send a GET request to a URI that specifies the bucket and includes the location. The S3 service responds with an HTTP 200 message containing an XML document that describes the bucket's location.

To list the S3 buckets that belong to your account, you send a GET request to the S3 service itself. The service will respond with an HTTP 200 message containing an XML document that lists the buckets associated with your S3 account in alphabetical order.

Here is an XML document returned by the operation:

```
<ListAllMyBucketsResult xmlns='http://s3.amazonaws.com/doc/2006-03-01/'>
  <Owner>
    <ID>1a2b3c4d5e6f1a2b3c4d5e6f1a2b3c4d5e6f1a2b3c4d5e6f1a2b3c4d5e6f1a2b</ID>
    <DisplayName>jamesmurty</DisplayName>
  </Owner>
  <Buckets>
    <Bucket>
      <Name>my-bucket</Name>
      <CreationDate>2006-12-13T21:21:14.000Z</CreationDate>
```

```
</Bucket>
</Buckets>
</ListAllMyBucketsResult>
```

To delete a bucket, you send a DELETE request to a URI specifying the name of the bucket. When the request is successful, the S3 service responds with an empty HTTP 204 response message. You cannot delete a bucket unless it is empty.

**S3 objects** are resources that store data. They are somewhat similar to the files in a standard computer system. An object can contain up to 5 GB of data, or it can be entirely empty. An object can store two types of information: data and metadata. The data stored by an object is its main content, such as a photo or text document. In addition to the data content, an object can store metadata that provides further information about the object, such as when it was created and the type of data it contains. You can store your own metadata information when you create or replace an object.

Each object resource in S3 can have access control permissions applied to it, allowing you to keep the object private, or to make it available to other S3 users or the general public. Each object in S3 is identified by a unique name, known as its key, which uniquely identifies it within a bucket. Object keys must not be longer than 1,024 bytes when encoded as UTF-8, and they can contain almost any characters, including spaces and punctuation. Objects are similar to files, so it makes sense to use obvious names for your objects, as you would for a file, such as *My Birthday Cake.jpg*.

One major difference between the S3 storage model and the average computer file system is that S3 has no notion of a hierarchical folder or directory structure. S3 buckets contain objects—that is the beginning and end of the hierarchy imposed by the storage model. If you wish to impose a hierarchical structure for your objects in S3 to help organize and search them, you must construct this hierarchy yourself using the flexible naming capabilities of object keys. You can do this by choosing a special character or string to mark the boundaries between components of a hierarchical path and by storing your objects with key names that describe their full path in the hierarchy.

To create an object in S3, you send a PUT request containing the object's data and metadata to the service, with a URI specifying the key name of the object and the bucket it will be stored in. The object's data content is provided as the body of the request, while the metadata is provided as request headers. When the object is successfully stored, S3 will return an HTTP 200 response message.

There are two ways to retrieve information about an object from S3: using a GET request or a HEAD request. To retrieve all of an object's data, including both its content and metadata, you send a GET request to the service with a URI specifying the bucket the object is stored in and its key name. The response to a GET request will contain the object's data in the response body and its metadata in the response headers. To retrieve only an object's metadata and not its contents, you send a HEAD request to the service instead. The response to a HEAD request includes the metadata headers but contains no body. Both GET and HEAD requests will return an HTTP 200 status code when they are successful.

You may wonder why you would ever use the HEAD method, when you can simply use the GET method to retrieve all of an object's data and ignore the response body, if you are not interested in the object's contents. It is worthwhile to use HEAD requests when you are only interested in an object's metadata, because your client will not have to maintain an open network connection any longer than necessary or manually close the connection to discard the unwanted response-body data.

To obtain a listing of the objects you have stored in a bucket, you send a GET request to a URI that specifies the bucket resource. S3 will reply with an HTTP 200 response message that contains an XML document in the response body. This XML object-listing document contains an inventory of the objects in a bucket, including important information about each object, such as its key name, size, last modification date, and the MD5 hash value of its data.

Here is an XML document returned by the operation:

```
<ListBucketResult xmlns='http://s3.amazonaws.com/doc/2006-03-01/'>
  <Name>my-bucket</Name>
  <Prefix/>
```

```

<Marker/>
<MaxKeys>1000</MaxKeys>
<IsTruncated>false</IsTruncated>
<Contents>
  <Key>Metadata.txt</Key>
  <LastModified>2007-11-06T02:52:41.000Z</LastModified>
  <ETag>"2d6b5bb20f322240448c5a25924ab4f5"</ETag>
  <Size>16</Size>
  <Owner>
    <ID>1a2b3c4d5e6f1a2b3c4d5e6f1a2b3c4d5e6f1a2b3c4d5e6f1a2b3c4d5e6f1a2b</ID>
      <DisplayName>jamesmurty</DisplayName>
    </Owner>
    <StorageClass>STANDARD</StorageClass>
  </Contents>
<Contents>
  <Key>WebPage.html</Key>
  <LastModified>2007-11-06T02:51:35.000Z</LastModified>
  <ETag>"ac3dcff8205758dc98c2d72a2ae8a3a5"</ETag>
  <Size>29</Size>
  <Owner>
    <ID>1a2b3c4d5e6f1a2b3c4d5e6f1a2b3c4d5e6f1a2b3c4d5e6f1a2b3c4d5e6f1a2b</ID>
      <DisplayName>jamesmurty</DisplayName>
    </Owner>
    <StorageClass>STANDARD</StorageClass>
  </Contents>
</ListBucketResult>

```

This document has a complex structure and contains much more information than just an inventory of our objects. Let us work through the elements in this document structure by dividing them into three categories: object details, truncated listings, and searching. In an object-listing document, the information that is most apparent, and most immediately useful, is the actual list of objects. In the ListBucketResult document structure, objects are listed as a set of Contents elements, which contain objects sorted alphabetically by their key name. Each Contents element includes a number of useful details about the object it represents. Inside each Contents element we can see important details about the object, including its key name (Key), a timestamp describing when it was created or last modified (LastModified), a hex-encoded MD5 hash value of the object's data content (ETag), and the size of the object in bytes (Size). In addition to these object details, the listing includes an Owner element that identifies the owner or creator of the object. Unless you are listing the contents of a bucket owned by someone else, or you have permitted someone else to create objects in your bucket, you will be the owner of all the objects in your bucket. Finally, the Contents element includes a StorageClass designation for each object. This is not a meaningful piece of information, because the storage class of objects is always Standard.

S3's object-listing API provides basic searching functionality to find objects based on their key names. When you perform object-listings, you can take advantage of this structure and list only those objects that occupy a specific place in the hierarchy.

The listing API accepts two parameters that allow you to search for objects with specific names: prefix and delimiter. If you include the prefix parameter value in your object-listing request, only those objects with key names that start with the prefix string will be listed. If you include the delimiter parameter, any objects with key names that contain the delimiter string will be listed in a separate part of the XML document, as if they were directory names. If you use both of these parameters at the same time, you can navigate through the hierarchy represented in your object key names like you would through the subdirectories in a standard file system.

These parameters are not easy to describe, so let us look at some examples to make things clearer. Imagine that you have stored a number of images in a bucket with key names that represent a hierarchy, like this:

```

MyPictures/2005/image1.jpg
MyPictures/2006/image2.jpg

```

To list only the images that date from 2005, you could perform a listing with the prefix parameter set to MyPictures/2005/, and only the object named *MyPictures/2005/image1.jpg* would be listed. To find out how many year-based subdirectories are inside this hierarchy, you would perform a listing with the prefix value MyPictures/ and the delimiter value / (a forward slash). With these two parameters, the listing will not contain any

objects at all, because the delimiter character is present after the prefix string in all the object keys. Instead, the XML listing document will include a set of CommonPrefixes elements containing a Prefix value corresponding to each unique object key name that includes both the prefix and the delimiter.

```
<CommonPrefixes>
  <Prefix>MyPictures/2005/</Prefix>
</CommonPrefixes>
<CommonPrefixes>
  <Prefix>MyPictures/2006/</Prefix>
</CommonPrefixes>
```

As you can see, the CommonPrefixes resemble a subdirectory listing of the *MyDocuments* directory. These CommonPrefixes strings are very useful for navigating hierarchies, because you can apply them as prefix parameter values to follow-up requests to drill down into the contents of each of the simulated subdirectories.

## S3 Applications

S3 can be used in a number of ways to meet many different storage needs. You can use it as a basic online file repository for backing up files, for web site hosting, as the basis for a network-mounted filesystem, or as a distribution network. In this chapter we discuss how you can use S3 to fulfill some common tasks by taking advantage of some of the available tools, software libraries, or third-party services. The S3 developer ecosystem is very active, and much of the third-party software available is open source and free. In many cases you can achieve a great deal without having to write your own code, and even if you have specific needs not yet met by existing software, there are mature libraries available in a range of languages that you can use to build your own solution.

**Share Large Files.** A very simple application for S3 involves using the service as a repository for sharing files that are too large to include in an email. There are a number of online services already available to do this job, but many charge monthly subscription fees if you need to share very large files; with S3 you can do this yourself at little cost. To share a file with your friends or colleagues, you will need to upload the file to S3 and send a URI link to the S3 object in an email. Because your files may contain private information, we will keep them private by generating a signed Universal Resource Identifier (URI) link to the object so that only the people who receive the link from you can access it. An advantage of using a signed URI is that you can choose how long the link will remain valid.

**S3 Filesystem with ElasticDrive.** One of the most interesting potential uses for S3 is as an unlimited data store on top of which other filesystem interface abstractions can be built. Some of these tools are designed to make S3 storage resources accessible to existing network-based tools that do not recognize S3—for example, as a File Transfer Protocol (FTP) or a Web-based Distributed Authoring and Versioning (WebDAV) service—and others aim to make the storage space in S3 available as a lower-level filesystem resource. Both of these approaches provide benefits, but it is the S3-based filesystem approach that we will concentrate on in this section, because it presents the most interesting possibilities. It also presents the most difficult challenges. If it proves to be feasible to build whole filesystems on top of S3, many of the service's limitations could be overcome in a very elegant way. Rather than having to use specialized S3 tools to access your storage space, you can make the service look and behave like a standard disk drive that stores data reliably in the cloud behind the scenes. On your computer you could copy files to and from this disk, even rename and rearrange them. In the background the changes you make would automatically be translated into API requests and stored in S3. This approach also makes it possible to use advanced disk management protocols, like RAID mirroring, to automatically manage synchronization between a local file system and S3, effectively giving you an effortless, online backup of all your files.

**Mediated Access to S3 with JetS3t.** The S3 service can be a very effective platform for sharing information, when its simple access control mechanisms meet your needs; but the level of control possible with the service's ACL settings may not always be sufficient. Some scenarios are difficult or impossible to achieve with ACL settings alone, such as if you wish to make your S3 storage available to your customers or colleagues to use when they do not have their own AWS account. In such cases you may need to provide your own intermediate service to mediate access to your S3 storage. The JetS3t Java library to mediate third-party access to your S3 storage. These tools include a client-side application, for interacting with S3 to upload and download files, and a server-side Gatekeeper component that decides whether the client, or user, should be authorized to perform these operations.

### 3.3.23.3 EC2: Elastic Compute Cloud

EC2 provides an environment for running virtual servers on demand. You can manage each virtual server like a physical machine, installing the software you need and configuring it to work the way you want, or you can use preprepared servers created by third parties. The service allows you to create a resizable pool of servers for handling computing tasks. You can start as many virtual servers as necessary to perform a task, increase or decrease the number of servers as demand rises and falls, and stop them all when the task is finished. You pay only for the computing resources you use. In addition to scaling out by increasing the number of servers that will work on a task, you can scale your computing power up or down by using more or less powerful virtual server types. The EC2 service comprises three key components:

- *Instances.* EC2 instances are the virtual machines that run in the EC2 environment and perform computing tasks that would typically be done by physical servers.
- *Environment.* Instances run in the EC2 environment, which provides configurable access control, contextual data, and other information that instances need to do their work.
- *Amazon Machine Images.* Amazon Machine Images (AMIs) are files that capture a complete snapshot of an EC2 instance at a point in time, including its software, configuration, and potentially even its data. These images serve as the boot disk for the instances you launch.

An EC2 virtual machine **instance** is a Linux computer system to which you have full root (administrator) access. The instances run in a Xen virtual environment. This means that the underlying computer hardware is virtualized and can be tailored to give performance within defined specifications, regardless of what real physical hardware Amazon is using in their data centers. Despite the fact your instances do not run on hardware in the usual sense, they are configured to provide a well-defined amount of computing power.

To provide a baseline guide to the computing capacity you can expect from an EC2 instance, Amazon defines their own measure of processing power, called an *EC2 Compute Unit*. This measure is based on several benchmarks set by Amazon to ensure that the performance of EC2 instances remains consistent and predictable over time. As a reference point, you can expect an instance with a rating of 1 EC2 compute unit to provide the same CPU capacity as a physical machine with a 1.0 to 1.2 GHz AMD Opteron processor, circa 2007.

EC2 offers a choice of three virtual machine instance types that offer different levels of performance and resourcing, and based on different platforms, with corresponding differences in pricing. The three instance types are referred to as small, large, and extra-large. Table 3.30 lists the specifications provided by the three instance types. The processing power you can expect from each instance type is defined in terms of EC2 compute units (ECUs). The measure of I/O performance is more nebulous, and is merely divided into two broad categories: "moderate" and "high." Unless you explicitly choose otherwise when you launch your instances, the service will launch the small instance type as a default.

**Table 3.30. EC2 instance types**

Resource	Small	Large	Extra Large
Platform	32-bit x86	64-bit x86	64-bit x86
CPU rating	1 ECU (1 virtual core)	4 ECUs (2 virtual cores, 2 ECUs each)	8 ECUs (4 virtual cores of 2 ECUs each)
Memory (RAM)	1.7 GB	7.5 GB	15 GB
Storage (ephemeral)	150 GB	840 GB (two 420 GB partitions)	1680 GB (four 420 GB partitions)
Storage (root partition)	10 GB	10 GB	10 GB
I/O Performance	Moderate	High	High
Instance Type Name	m1.small	m1.large	m1.xlarge

An EC2 instance is based on an **AMI** that captures the root filesystem of an instance in a series of files. When you launch an instance, it boots from the image's filesystem and runs with the software, configuration settings, and data that were stored in the AMI. You can log in to a running instance as the root user and customize it for your own purposes by installing and configuring any additional software that is compatible with the instance's Linux kernel. Any changes you make to the instance can then be bundled up into a new AMI to be used as the starting point the next time you start an instance. You can even create your own AMI from scratch using your preferred Linux distribution. AMIs are stored in Amazon's S3 service and must be registered to be recognized by EC2. The

images may be registered for private use only, in which case only you may run the AMIs you have created; or they may be shared with other EC2 users. Images can be shared with specific users or made completely public and available to all EC2 account holders.

In addition to sharing your AMIs, you can also rent them out to other users for a fee. If you create an image that is of value to others, you can register the image as a Paid AMI through Amazon's DevPay system and arrange to have a product code associated with it. EC2 uses the product code to recognize when your AMI is run and by whom, at which point the service will charge the AMI user the additional usage fee premium you specify on top of Amazon's hourly fees. Users who wish to run your AMI must first go through a sign-up process to provide Amazon with their credit card details, so they can be billed for using the image. Amazon collects the hourly premium on your behalf and passes the money on to you. Product codes can also be used for other purposes, such as to identify images you have sold to a user with a support agreement.

The EC2 **environment** provides instances with a range of services, such as access control mechanisms, a network firewall, network address allocation, and ephemeral storage volumes. EC2 provides support for public and private key-based login access to instances, using the widely-used Secure Shell program (SSH). This mechanism is based on keypairs, a topic we discuss in detail in this chapter. Keypairs comprise a private key that you receive from Amazon and store on your client computers and a corresponding public key component that is stored within the EC2 environment. When you start an EC2 instance, you can choose to have the environment make one or more of your public keys available to the instance. If the instance is suitably configured, it will use this public key as half of the access credentials necessary to log in, so that only someone with the corresponding private key can access the instance.

The EC2 environment also provides a basic firewall mechanism that allows you to limit the network traffic that will reach your system. Firewall settings are configured using security groups that define rules specifying which incoming network connections will be allowed. These connection rules can impose limits based on the transport protocol, the port number, and the source IP address of the network traffic. Security groups, which we will discuss shortly, can be combined together to produce a cumulative set of rules, and these firewall rules can be modified while your instances are running.

When you launch an instance, the EC2 environment assigns it dynamic internal and external network addresses. These addresses will refer to your instance during its lifetime, and when the instance is terminated, the addresses will be reclaimed by the service and reused. There is no mechanism in EC2 to maintain a single, static network address for any of your instances.

Each instance is given storage space in the EC2 environment. The amount of space allocated to the instance depends on the instance type. This storage space is ephemeral, meaning that it is only allocated to your instance for its lifetime. Any data written to an instance's data storage resources will be lost when the instance is terminated. It is the responsibility of you, the instance owner, to implement a persistence strategy if you need your data to outlive the instance on which it is stored.

EC2 account holders are billed monthly for their usage of the service. The service charges are based on two aspects of your EC2 usage: the number of instance hours used and the volume of data transferred to or from your EC2 instances.

When you launch an EC2 instance, the first thing you will generally want to do is log in to it over a Secure Shell connection to run programs on the instance and control what it is doing. When your EC2 account was created, it will have been provided with a security group named default. This group is preconfigured to allow your EC2 instances to communicate with each other but it does not allow network access from outside the EC2 environment. If you were to start an EC2 instance with the default rules in the default group, there would be no way for you to access the instance from your own computer over the Internet. To permit network traffic from outside EC2 to reach an instance, we must create security group rules that allow IP traffic to pass through the EC2 firewall. The security group rules that permit IP network traffic from outside or inside the EC2 environment are called CIDR rules. CIDR stands for Classless Interdomain Routing, and permit incoming Internet Protocol traffic to reach your EC2 instances. You can control what kind of traffic is permitted based on criteria such as the address the traffic originated from, the protocol it is using, and the target port it is directed to. To launch an EC2 instance, we first need an AMI to launch the image from. To start an AMI instance, we need to know the identifier value assigned to the AMI in EC2. The identifier we need is distinct from the location of the AMI's manifest, and it can potentially change over time if the AMI file is updated or reregistered. In fact, the ID value for this AMI may well

be different by the time you read this book. To launch an instance from an AMI, we must first look up its identifier value.

When you run an instance in Amazon Elastic Compute Cloud (EC2), you gain all the power and the responsibility of a server administrator. You can install any software you want and configure your instances as you please, but you must ensure that they work correctly, are properly secured, are resistant to faults, and do not violate Amazon's acceptable use provisions.

EC2 instances are based on a Xen-compatible Linux kernel compiled with the GNU C Compiler (GCC) version 4.0. Unlike standard Linux machines, it is not possible in EC2 to use an alternative kernel to the one provided by Amazon, because the Xen software provides the kernel, rather than reading it from the boot disk. Therefore any software you use in your instance must be compatible with the kernel provided; if you wish to use additional modules, these must be loaded dynamically, rather than compiled into the kernel.

## EC2 Applications

You can use the virtual servers provided by the EC2 service to do most things a physical server can do, from hosting web sites or applications to creating clusters of servers for on-demand processing of large data sets.

**Dynamic DNS.** If you run a public-facing service on EC2, such as a web site or web application, you will want to make your instance accessible via a user-friendly domain name that your users can remember. With standard servers you would achieve this goal by purchasing a domain name and configuring the Domain Name System (DNS) settings for that domain to refer to your server's IP address. However, this approach is only really workable if your server has a static IP address that does not change over time. EC2 does not allow network addresses to be statically assigned to instances. When you start an EC2 instance, the virtual machine is assigned IP and DNS addresses that will only refer to your instance for as long as it is running. If you terminate your instance, or if the instance fails, it will be assigned a different address when it is launched again. The lack of static addresses in EC2 makes it very difficult to use the standard DNS system effectively without risking significant downtime for your service, because regardless of how quickly you can get a new instance up and running, any address changes will take time to propagate through the DNS system. And until the instance's new address is fully propagated, your users will be unable to access your server. A partial workaround for the lack of static addresses is to use a dynamic DNS service to associate your domain name with your EC2 instance instead of standard DNS. Dynamic DNS services are designed for situations in which a server's address changes every so often, and they will propagate address changes to the public much more quickly than standard DNS. These services only provide a partial workaround, because it still takes time for changes to propagate from dynamic DNS services; and there can be other complications, such as client software that caches old addresses. However, unless and until Amazon provides static addresses in EC2, these services offer the easiest and cheapest way to make your instance accessible via an unchanging domain name.

**On-Demand VPN Server with OpenVPN.** A key advantage of EC2 is that you can start and stop server instances as you need them and only pay for the time the server is running. This capability is most often useful for increasing and decreasing the number of servers you have running in response to changing demands on a web application. In this application, however, we will use an on-demand server in a very different situation. We will demonstrate how to set up an EC2 instance to run a Virtual Private Network (VPN) server that you can use to secure your network traffic when you access the Internet over an untrusted network. It is becoming increasingly common for people to access the Internet through public access points, such as WiFi hotspots, wired networks provided by hotels, or the internal networks of companies you may be visiting. The availability of Internet access points is fantastic when you are away from work or home, but this convenience comes with risks when you cannot be sure that the network is secure and safe from snooping. The best way to protect your data when using an untrusted network is to use a VPN to encrypt it. In this application, we will create an EC2 instance that runs the open-source VPN server OpenVPN (<http://openvpn.net/>). OpenVPN is a freely-available, powerful, and highly configurable VPN server; best of all, it is relatively simple to set up compared to most other VPN servers. We will configure the server to use a secret key such that only you, the owner of the key, can connect to it. Once we have configured our instance, it will allow a client computer to connect over a secure channel, and it will relay all network traffic to the public Internet on behalf of the client. Whenever you need to access the Internet over an untrusted network, you can fire up this instance and create your own personal VPN to protect your network traffic.

### **3.3.23 Google Web Services**

Google is one of the most popular search engines around because it provides a superior number of hits. Of course, more hits don't translate into better data. The search engine is also good at providing valid information through the use of indexing and filtering so long as you specify the search criteria clearly. Given the number of ways that the Google Advanced Search ([http://www.google.com/advanced\\_search](http://www.google.com/advanced_search)) helps you look for information, providing clear direction can be overwhelming to some. The flexibility provided by the interface is part of Google's charm, however, and the reason many power users prefer Google. If you can't describe a search using this interface, you might not know what you're looking for.

#### **3.3.23.1 Search services**

Google Web Services is a means of accessing Google without going to the Web site and performing a search manually. This Web service provides essential services by helping you automate the search process and presenting data in the form that you need, rather than in the form that Google thinks you need. From a Google Web Services perspective, you request information based on any of a number of search criteria. Google supports a number of search techniques and not every technique works well for every kind of search. For now, just think of the search criteria as a form of request. The request defines the kind of information you want to know and how detailed that information will be. Google Web Services returns the information you request in a standardized format.

Google's database *schema* specifies the format of the information. A schema defines the organization of information in a database. Fortunately, the format of the data returned by Google is relatively simple. You only have to consider a few return types. However, the content of the return data is a different story.

A Web service also performs some type of useful work. The useful work might be something as simple as interpreting your request, calculating the answer, and sending the result back. In the case of Google Web Services, the Web service accepts your request in the form of a search request, interacts with the database through a search engine to obtain the information you requested, and sends the information back to you. The search can take various forms. For example, you don't have to search all Web sites—you can concentrate on just one. You might want to look for pictures, rather than text, and might only have an interest in newsgroups.

The final consideration for a Web service (at least from the Web service user perspective) is that it executes on the remote machine, not on your machine. In short, this means you're using resources on that other machine with the permission of the machine's owner. The remote machine can set requirements for using the Web service, as well as require you to perform specific setup and security checks as part of your request. In the case of Google Web Services, you need to obtain this permission by requesting a license. You also need to download the Google Web Services Kit to ensure you follow the terms of the licensing agreement.

For example of Web services from other vendors, Microsoft supports the MapPoint Web Service (<http://www.microsoft.com/mappoint/net/>). You can search for companies that offer Web services using the Web Services Finder page at <http://www.15seconds.com/WebService/>. In some cases, you'll need to use a specialty Web service list such as the one at <http://www.flash-db.com/services/>. The Web services on this site are special because many of them perform one task well, such as providing you with a location based on a domain name.

Web services are actually quite easy to understand if you look at them in a way that relates the task to everyday occurrences. For example, you might compare the operation of a Web service to making a withdrawal at the bank—the process really is the same. The one thing to remember is that the process a Web service uses to perform a task is always the same. No matter what technology you use to make a request or receive a response, the steps are still the same. Here are the steps that most Web services (including Google WS), use to complete a transaction.

1     *The client discovers the WS.* During the act of discovery, the client might do things like download a file that tells how to interact with the Web service. This step is the same as someone walking into the bank. The person knows the bank exists and the bank teller might have noticed the person. The bank posts the rules for making a withdrawal or the teller might help a first-time customer understand the rules.

2     *The client makes a request based on the rules delivered during the discovery phase.* The rules might specify that the request has to appear in a certain form, and the client must provide specific data. This step is the same as the person walking up to the teller's window with a withdrawal request. The request must contain the person's account number, the amount they wish to withdraw, and other identifying information. The bank specifies the format of the request and the information it must contain.

3     *The server might ask the client for credentials depending on the openness of the Web service.* Google Web Services is public but still requires that you supply a developer license (account) number as identification. This step is the same as the bank teller asking you for a driver's license or other form of identification before honoring your withdrawal request.

4     *The WS performs the work required to honor your request.* In most cases, the WS accesses a database for information, it could enter an order, and it might even provide some level of formatting information about the original information (such as the typeface used for a document). Google WSs performs a number of tasks depending on the request you make. The easiest request is a general search, but you can also perform checks such as making a spelling check. This step equates to the bank teller getting the money from the drawer and counting it.

5     *The WS sends the data to the client.* The content of the information depends on the Web service. Google Web Services provides data in a very specific format based on the content of the associated database and the nature of the request. This step equates to the teller handing the person their money. In general, the teller orders the money in a specific way and counts it out to the person, rather than simply handing the money over.

6     *The client logs out of the WS or the WS disconnects the client after some period of inactivity.* This step equates to the person leaving the bank, money in hand. If the person doesn't leave the bank (they just hang out in the lobby), you can be sure that someone will ask them to leave.

7     *The client does something with the data it receives.* In many cases, it formats the data and presents it on screen for the user. This step equates to the person spending the money they receive from the bank.

From a client perspective, the type of device you use to access the Web service determines the access speed, as well as what you can do with the data once you receive it. Although a PDA such as the Pocket PC can access Google Web Services just fine, you wouldn't want to use it to perform detailed searches or attempt complex activities such as converting data to another language. About the best you can hope for is to perform simple research. On the other hand, a desktop or laptop machine has all of the processing power, screen real estate, and functionality to perform any task. Google Web Services hasn't changed, but the capability of the client has.

The Pocket PC provides additional functionality and features that make it a better target for some types of applications than devices such as the Palm. On the other hand, most Palm devices are much easier to carry and cost less than the Pocket PC. This book examines the entire range of mobile devices to ensure you understand the limitations of using a specific device to access Google Web Services.

Google Web Services also has some usage requirements and these requirements might change the way that you use your client. For example, according to the license agreement you can't make more than 1,000 requests per day—at least, not without special permission. The request limitation ensures the Google servers won't become overloaded, but they also mean you must provide some type of monitoring in your application to prevent abuse of the licensing terms. If you violate the licensing terms, Google Web Services simply denies your request. Often, you can get around the licensing requirements for a Web service by using smart programming techniques. Google doesn't require that you refresh the information you receive at any specific interval. You determine when the information you receive is too old. Using good caching techniques means that you can create applications that are lightning fast, unless the request is new or the data is old. Although it seems as if a 1,000-request limit could cause problems, you can usually satisfy far more than 1,000 requests per day by using smart data caching.

One of the most common uses of Google is performing research. Research searches normally begin based on keywords. The problem is that some keywords are ambiguous enough that the resulting data isn't meaningful. Google Web Services can help because you can automate multiple searches to locate specific information. For example, say you want to use a particular class or you have a special need in the application. Performing the search manually could require multiple trips to Google. However, using Web services means you could enter the criteria once and let the application make the multiple searches for you. Even given the speed of an automatic search, you might wonder whether it's worth the effort of using Google Web Services. However, an application can do something that a manual search can't. Once the application returns from the search, it could store the results. The application would continue with each search scenario until it finished. Then the application could analyze the various returns and create a list of most likely sites based on the results.

Expansion searches help you locate all available information on a topic by playing to the features that Google provides. For example, the order of search terms is important in the way that Google interprets a search. In addition, if you work in an acronym-laden field, expanding the acronyms is important to locate all sources of information on a given topic. Manual expansion searches become cumbersome for a number of reasons. Repetition is one of the main causes, but there are others such as entry errors and result interpretation. You have to

provide enough keywords to make a search specific, but each keyword adds an order of complexity to the expansion search. Google Web Services steps in by letting you perform an expansion search automatically using code. You supply the four keywords—the code does the rest. By comparing the results of each expansion search, you can come up with an optimal group of sites. For example, you could verify that the site appears in every expansion search return, which tends to reduce the false positives. You can also rate the sites based on the number of times they appear and their position in the list. Although it's possible to perform this kind of data manipulation using a manual search, no one would want to do it.

Some Web sites don't provide a search engine. The site might be too small to support a search feature or hosted so the developer doesn't have access to the server's search feature. In other cases, a site does provide a search engine, but the search engine doesn't work nearly as well as Google's. You may find that the search engine fails to produce the desired results, even when you know the information exists. In both cases, you can create a site-specific search using Google Web Services. You can perform this kind of search manually. In fact, it's not even all that time consuming. However, remembering the information you have to provide in an URL or going to Google's advanced search site every time you want to perform the search is a headache. Using Web services lets you store all of the static settings—the ones that won't change—so that all you need to know is what keywords you want to enter for that site. A site-specific search is all about convenience. Using this technique makes it easier to get the information you need without a lot of effort. One way to use this technique is to create a search setup for your personal Web site. Many Web sites owned by individuals or the self-employed appear on hosted sites, making it impossible to add search capability with any ease. A Google Web Services application can make it easy to add a professional search service to your site, making it a lot more attractive to anyone who visits. Another way to use this technique is to create custom search Web pages. I built one for my personal use that includes links to all my favorite coding sites. All I do now is select the site I want to search, add a few keywords, and Google Web Services takes care of all the hard work for me. Not only am I more productive, but I can stay focused on the task at hand—finding sample code. I can even make searches of multiple sites with a single click.

What do you really know about a Web site before you visit it? This question takes many people by surprise because they have to admit that they really don't know anything about the site. However, visiting a site implies that you're willing to open yourself to anything the site can provide within the limitations of your browser. A site that contains pornographic material or a virus when you're conducting legitimate research on parts of the human anatomy is an unwelcome surprise that you could avoid. Google provides a number of searches you can use to verify the usefulness of a Web site before you visit it. For example, you can begin by looking for keywords in the snippet and site summary that Google provides. An examination of the links for the site, along with the Web information it provides is revealing. You can also conduct a related links search to see how the site connects to the rest of the Internet. If you're truly uncertain about the usefulness of the site, you can view a cached version of the page. The cached version does contain old data, but it can help you check for objectionable terms or content without exposing yourself to as much risk. The point is that with the security problems that users face today, they need a better way to assess the risk of visiting a particular site online. A fact-finding search is very useful in keeping some types of Internet risks at bay. Unfortunately, few users are going to take the time to perform such fact-finding before visiting a site. It's simply easier to click on the URL and go there. However, you could build a Google Web Services application that would display the search results and assess the potential of a Web site before the user visits there, while maintaining single click efficiency. When the user clicks a link, your application can check the site in the background and verify that it's reasonably safe. What the user sees is the normal sequence of events that take place when they click the link.

The Internet is constantly changing. In fact, it changes so fast sometimes that it's hard to keep all of the links updated. Anyone who spends any amount of time researching information online knows that even the links Google provides get outdated. However, seeing an error message, page not found, when you click that link isn't the end of the road. You can request cached data from Google. The cached information is old in many cases, but at least it's available and you can use it for whatever you need. Like many other kinds of Google searches, you can perform a cache search manually. However, you have to perform multiple keystrokes to perform the search, assuming you remember to do it. Many people simply move on to the next site without thinking when they reach an error message. Cached data searches can be very helpful, especially during research.

A Google Web Services application can reduce the problems of the dead link. It could begin by searching for the site. If the site isn't available, the application can move on to the cached page. When Google doesn't provide a cached page (a rarity), the application can move on to related links. Even if these techniques fail, the application could use some kinds of regressive searching. A regression search is one in which you begin with the result data

and look for the information used to create the results. The point is the user wouldn't see an error message—a page of some kind would display and the user would then make the decision on the value of that page.

The spelling check is one of the few Google WSs tasks that you can't perform manually. You send a string (up to 2,048 characters long) to Google WS. The WS checks the string for spelling errors and sends the corrected string back to you. At first, you might wonder how you would use this service. After all, it's easy to find a local spell checker that won't use one or more of the 1,000 calls that Google allots to each developer per day. The answer is that you wouldn't use this service personally in most cases. However, if you're running a Web site that requests text input from users, you can use the spell checker to validate their work. Because the data you receive as input from the user contains fewer errors, you'll also end up doing less work. For example, any database you use to maintain the user input will have fewer errors, so you'll spend less time looking for errant records.

Google Web Services is no different from any other Web service when it comes to output. You'll provide input, receive raw data, manipulate that data in some way, and view the output—the result of everything you have done with the Web service. A Web service has no concept of data type when it comes to the data itself. Every data transfer is text. The XML used to transfer the data does include type information, but of the sort that's normally associated with database fields, which means you have to know the field names to make an interpretation. For example, you might receive data in a message like the one shown here.

```
<item xsi:type="ns1:ResultElement">
  <cachedSize xsi:type="xsd:string">12k</cachedSize>
  <hostName xsi:type="xsd:string" />
  <snippet xsi:type="xsd:string">
    <b>...</b> some text <b>highlight</b>) more text <b>...</b>
  </snippet>
  <directoryCategory xsi:type="ns1:DirectoryCategory">
    <specialEncoding xsi:type="xsd:string" />
    <fullViewableName xsi:type="xsd:string" />
  </directoryCategory>
  <relatedInformationPresent xsi:type="xsd:boolean">
    True
  </relatedInformationPresent>
  <directoryTitle xsi:type="xsd:string" />
  <summary xsi:type="xsd:string" />
  <URL xsi:type="xsd:string"> http://www.mwt.net/ </URL>
  <title xsi:type="xsd:string"><b>DataCon Services</b></title>
</item>
```

Google Web Services sends all data as characters (as do all other Web services) and defines the data using tags (the words between the angle brackets) and attributes (extra information within the tag). For example, the line that contains `<URL xsi:type="xsd:string"> http://www.mwt.net/ </URL>` includes the `<URL>` tag that tells you that this value is `http://www.mwt.net/` and that the tag type is an `xsi:type="xsd:string"`. The tag tells you what kind of information this is. By knowing the Google database layout, you also know the data type and other information about the entry. However, the information you receive from Google is still plain text. All of the tags and other information supplied in a request and response consume space. The file is larger than a text file with the same data because of all the tag information required. In addition, it's far more efficient to store many data types in their native format, rather than use characters. Consequently, Web service data suffers from bloat. The data uses more bandwidth than a binary message and consequently, you could experience performance problems. Because of this issue, you need to create efficient queries for your application that maximize data throughput despite the limitations of the XML format.

The results you obtain from Google are largely a matter of the input you provide in the form of a request. The query can become quite complex because even the order of the words makes a difference in the results you receive. Google must make this assumption because most people enter the words in the order they think about them, which is usually most important to least important. Consequently, if you always assume that your first query returns all possible results, you'll find Google Web Services disappointing.

The ranking of results you receive from Google Web Services is also unlikely to be the same as the ranking you need. Google sells keywords to make some sites turn up higher in the result list. In addition, Google often bases the site ranking on criteria that won't match your own, such as the number of times that a keyword appears. The bottom line is that the output you receive from Google is "raw" output—information that you haven't filtered or

organized in any way. One of the reasons to use Google Web Services is to enable you to perform tasks such as site ranking so the results appear in the order that's best for your organization.

### 2.3.23.2. Programming with Google App Engine

Google App Engine is a web application hosting service. By “web application,” we mean an application or service accessed over the Web, usually with a web browser: storefronts with shopping carts, social networking sites, multiplayer games, mobile applications, survey applications, project management, collaboration, publishing, and all of the other things we’re discovering are good uses for the Web. App Engine can serve traditional website content too, such as documents and images, but the environment is especially designed for real-time dynamic applications.

In particular, Google App Engine is designed to host applications with many simultaneous users. When an application can serve many simultaneous users without degrading performance, we say it *scales*. Applications written for App Engine scale automatically. As more people use the application, App Engine allocates more resources for the application and manages the use of those resources. The application itself does not need to know anything about the resources it is using.

Unlike traditional web hosting or self-managed servers, with Google App Engine, you only pay for the resources you use. These resources are measured down to the gigabyte, with no monthly fees or up-front charges. Billed resources include CPU usage, storage per month, incoming and outgoing bandwidth, and several resources specific to App Engine services. To help you get started, every developer gets a certain amount of resources for free, enough for small applications with low traffic. Google estimates that with the free resources, an app can accommodate about 5 million page views a month.

## The Runtime Environment

An App Engine application responds to web requests. A web request begins when a client, typically a user’s web browser, contacts the application with an HTTP request, such as to fetch a web page at a URL. When App Engine receives the request, it identifies the application from the domain name of the address, either an .appspot.com subdomain (provided for free with every app) or a subdomain of a custom domain name you have registered and set up with Google Apps. App Engine selects a server from many possible servers to handle the request, making its selection based on which server is most likely to provide a fast response. It then calls the application with the content of the HTTP request, receives the response data from the application, and returns the response to the client.

From the application’s perspective, the runtime environment springs into existence when the request handler begins, and disappears when it ends. App Engine provides at least two methods for storing data that persists between requests (discussed later), but these mechanisms live outside of the runtime environment. By not retaining state in the runtime environment between requests—or at least, by not expecting that state will be retained between requests—App Engine can distribute traffic among as many servers as it needs to give every request the same treatment, regardless of how much traffic it is handling at one time.

Application code cannot access the server on which it is running in the traditional sense. An application can read its own files from the filesystem, but it cannot write to files, and it cannot read files that belong to other applications. An application can see environment variables set by App Engine, but manipulations of these variables do not necessarily persist between requests. An application cannot access the networking facilities of the server hardware, though it can perform networking operations using services.

In short, each request lives in its own “sandbox.” This allows App Engine to handle a request with the server that would, in its estimation, provide the fastest response. There is no way to guarantee that the same server hardware will handle two requests, even if the requests come from the same client and arrive relatively quickly.

Sandboxing also allows App Engine to run multiple applications on the same server without the behavior of one application affecting another. In addition to limiting access to the operating system, the runtime environment also limits the amount of clock time, CPU use, and memory a single request can take. App Engine keeps these limits flexible, and applies stricter limits to applications that use up more resources to protect shared resources from “runaway” applications.

A request has up to 30 seconds to return a response to the client. While that may seem like a comfortably large amount for a web app, App Engine is optimized for applications that respond in less than a second. Also, if an application uses many CPU cycles, App Engine may slow it down so the app isn't hogging the processor on a machine serving multiple apps.

## **Creating an Application**

The App Engine development model is as simple as it gets:

1. Create the application.
2. Test the application on your own computer using the web server software included with the App Engine development kit.
3. Upload the finished application to App Engine.

In this section, we will walk through the process of creating a new application, testing it with the development server, registering a new application ID and setting up a domain name, and uploading the app to App Engine. We will look at some of the features of the Java software development kits (SDKs) and the App Engine Administration Console. We'll also discuss the workflow for developing and deploying an app.

We will take this opportunity to demonstrate a common pattern in web applications: managing user preferences data. This pattern uses several App Engine services and features.

## **Setting up the SDK**

All the tools and libraries you need to develop an application are included in the App Engine SDK. There are separate SDKs for Python and Java, each with features useful for developing with each language. The SDKs work on any platform, including Windows, Mac OS X, and Linux.

The Python and Java SDKs each include a web server that runs your app in a simulated runtime environment on your computer. The development server enforces the sandbox restrictions of the full runtime environment and simulates each of the App Engine services. You can start the development server and leave it running while you build your app, reloading pages in your browser to see your changes in effect. Both SDKs include a multifunction tool for interacting with the app running on App Engine. You use this tool to upload your app's code, static files, and configuration.

The tool can also manage datastore indexes, task queues, and scheduled tasks, and can download messages logged by the live application so you can analyze your app's traffic and behavior.

For Java developers, Google provides a plug-in for the Eclipse integrated development environment that implements a complete App Engine development workflow. The plug-in includes a template for creating new App Engine Java apps, as well as a debugging profile for running the app and the development web server in the Eclipse debugger. To deploy a project to App Engine, you just click a button on the Eclipse toolbar.

The SDK also includes cross-platform command-line tools that provide these features. You can use these tools from a command prompt, or otherwise integrate them into your development environment as you see fit.

## **Installing the Java SDK with the Google Plugin for Eclipse**

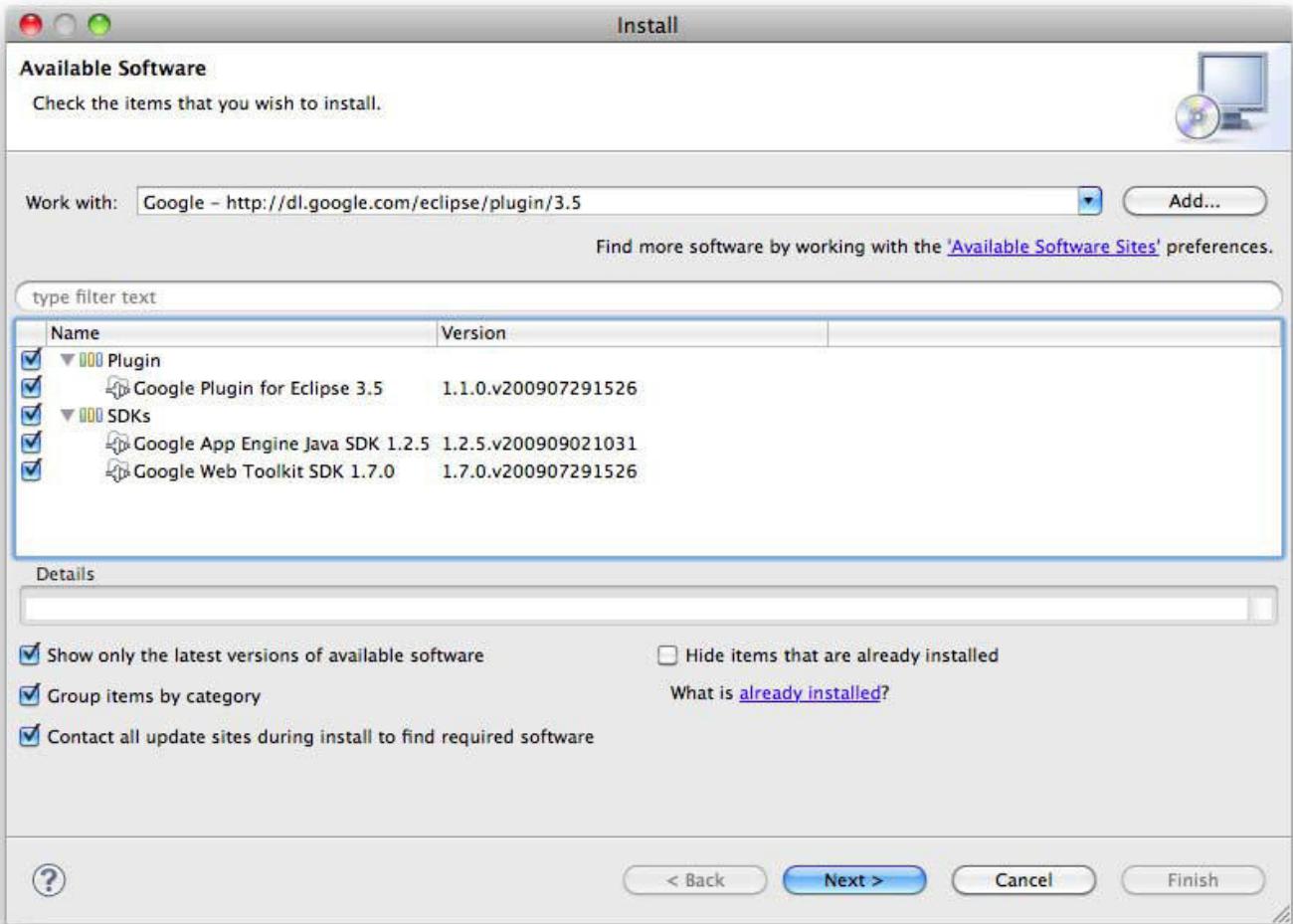
One of the easiest ways to develop App Engine applications in Java is to use the Eclipse IDE and the Google Plugin for Eclipse. The plug-in works with Eclipse 3.3 (Europa), Eclipse 3.4 (Ganymede), and Eclipse 3.5 (Galileo). You can get Eclipse for your platform for free at the Eclipse website: <http://www.eclipse.org/>

If you're getting Eclipse specifically for App Engine development, get the "Eclipse IDE for Java EE Developers" bundle. This bundle includes several useful components for developing web applications, including the Eclipse Web Tools Platform (WTP) package.

You can tell Eclipse to use the JDK you have installed in the Preferences window. In Eclipse 3.5, select Preferences (Windows and Linux, in the Window menu; Mac OS X, in the Eclipse menu). In the Java category, select "Installed JREs." If necessary, add the location of the SDK to the list, and make sure the checkbox is checked.

To install the App Engine Java SDK and the Google Plugin, use the software installation feature of Eclipse. In Eclipse 3.5, select Install New Software... from the Help menu, then type the following URL in the “Work with” field and click the Add... button: <http://dl.google.com/eclipse/plugin/3.5> (This URL does not work in a browser; it only works with the Eclipse software installer.)

In the dialog window that opens, enter “Google” for the name, then click OK. Two items are added to the list, one for the plug-in (“Plugin”) and a set for the App Engine and Google Web Toolkit SDKs (“SDKs”). The next Figure shows the Install Software window with the appropriate items selected.



**Fig. 3.80. The Eclipse 3.5 (Galileo) Install Software window, with the Google Plugin selected**

Check the boxes for these two items. Click the Next > button and follow the prompts. For more information on installing the Google Plugin for Eclipse, see the website for the plug-in: <http://code.google.com/eclipse/>

After installation, the Eclipse toolbar has three new buttons, as shown in next Figure.



**Fig. 3.81. The Eclipse 3.5 toolbar with the Google Plugin installed, with three new buttons: New Web Application Project, GWT Compile Project, and Deploy App Engine Project**

The plug-in adds several features to the Eclipse interface:

- The three buttons in the toolbar: New Web Application Project, GWT Compile Project, and Deploy App Engine Project
- A Web Application Project item under New in the File menu
- A Web Application debug profile, for running an app in the development web server under the Eclipse debugger

You can use Eclipse to develop your application, and to deploy it to App Engine. To use other features of the SDK, like downloading log data, you must use the commandline tools from the App Engine SDK. Eclipse installs the SDK in your Eclipse application directory, under `eclipse/plugins/`. The actual directory name depends on the specific version of the SDK installed, but it looks something like this:

This directory contains command-line tools in a subdirectory named *bin/*. In Mac OS X or Linux, you may need to change the permissions of these files to be executable in order to use the tools from the command line:

```
chmod 755 bin/*
```

You can add the *bin/* directory to your command path, but keep in mind that the path will change each time you update the SDK.

## Developing a Java App

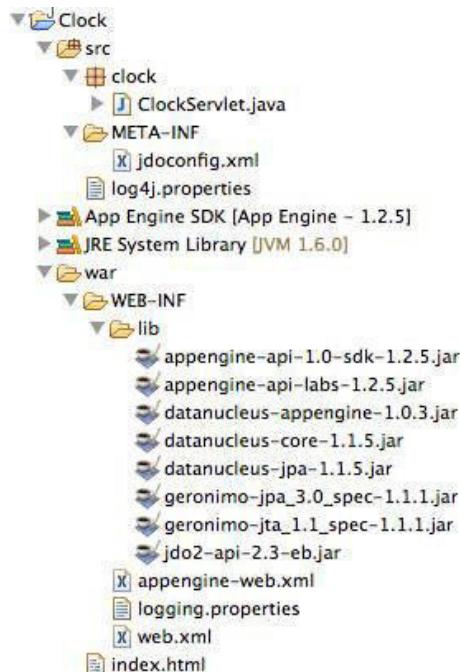
Java web applications for App Engine use the Java Servlet standard interface for interacting with the application server. An application consists of one or more classes that extend a servlet base class. Servlets are mapped to URLs using a standard configuration file called a “deployment descriptor.” When App Engine receives a request for a Java application, it determines which servlet class to use based on the URL and the deployment descriptor, instantiates the class, then calls an appropriate method on the servlet object.

All of the files for a Java application, including the compiled Java classes, configuration files, and static files, are organized in a standard directory structure called a Web Application Archive, or “WAR.” Everything in the WAR directory gets deployed to App Engine. It’s common to have your development workflow build the contents of the WAR from a set of source files, either using an automated build process or WAR-aware development tools.

If you are using the Eclipse IDE with the Google Plugin, you can create a new project using the Web Application wizard. From the File menu, select New, then Web Application Project. In the window that opens, enter a project name (such as *Clock*) and package name (such as *clock*).

Uncheck the “Use Google Web Toolkit” checkbox, and make sure the “Use Google App Engine” checkbox is checked. (If you leave the GWT checkbox checked, the new project will be created with GWT starter files.) Click Finish to create the project.

If you are not using the Google Plugin for Eclipse, you will need to create the directories and files another way. If you are already familiar with Java web development, you can use your existing tools and processes to produce the final WAR. For the rest of this section, we’ll assume the directory structure created by the Eclipse plug-in.



*Fig. 3.82. A Java project structure shown in Eclipse Package Explorer*

Figure 3.82 shows the project file structure, as depicted in the Eclipse Package Explorer. The project root directory (*Clock*) contains two major subdirectories: *src* and *war*. The *src/* directory contains all of the project’s class files in the usual Java package structure. With a package path of *clock*, Eclipse created source code for a

servlet class named *ClockServlet* in the file *clock/ClockServlet.java*. The *war/* directory contains the complete final contents of the application. Eclipse compiles source code from *src/* automatically and puts the compiled class files in *war/WEB-INF/classes/*, which is hidden from Eclipse's Package Explorer by default. Eclipse copies the contents of *src/META-INF/* to *war/WEB-INF/classes/META-INF/* automatically, as well. Everything else must be created in the *war/* directory in its intended location.

Let's start our clock application with a simple servlet that displays the current time. Open the file *src/clock/ClockServlet.java* for editing (creating it if necessary), and give it contents similar to Example 3.132.

**Example 3.132. A simple Java servlet**

```
package clock;
import java.io.IOException;
import java.io.PrintWriter;
import java.text.SimpleDateFormat;
import java.util.Date;
import java.util.SimpleTimeZone;
import javax.servlet.http.*;

@SuppressWarnings("serial")
public class ClockServlet extends HttpServlet {
    public void doGet(HttpServletRequest req, HttpServletResponse resp)
        throws IOException {
        SimpleDateFormat fmt = new SimpleDateFormat("yyyy-MM-dd hh:mm:ss.SSSSSS");
        fmt.setTimeZone(new SimpleTimeZone(0, ""));
        resp.setContentType("text/html");
        PrintWriter out = resp.getWriter();
        out.println("<p>The time is: " + fmt.format(new Date()) + "</p>");
    }
}
```

The servlet class extends *javax.servlet.http.HttpServlet*, and overrides methods for each of the HTTP methods it intends to support. This servlet overrides the *doGet()* method to handle HTTP GET requests. The server calls the method with an *HttpServletRequest* object and an *HttpServletResponse* object as parameters. The *HttpServletRequest* contains information about the request, such as the URL, form parameters, and cookies. The method prepares the response using methods on the *HttpServletResponse*, such as *setContentType()* and *getWriter()*. App Engine sends the response when the servlet method exits.

To tell App Engine to invoke this servlet for requests, we need a deployment descriptor. Open or create the file *war/WEB-INF/web.xml*, and give it contents similar to Example 3.133.

**Example 3.133. The web.xml file, also known as the deployment descriptor, mapping all URLs to ClockServlet**

```
<?xml version="1.0" encoding="utf-8"?>
<!DOCTYPE web-app PUBLIC
"-//Sun Microsystems, Inc.//DTD Web Application 2.3//EN"
"http://java.sun.com/dtd/web-app_2_3.dtd">

<web-app xmlns="http://java.sun.com/xml/ns/javaee" version="2.5">
    <servlet>
        <servlet-name>clock</servlet-name>
        <servlet-class>clock.ClockServlet</servlet-class>
    </servlet>
    <servlet-mapping>
        <servlet-name>clock</servlet-name>
        <url-pattern>/*</url-pattern>
    </servlet-mapping>
</web-app>
```

Eclipse may open this file in its XML “Design” view, a table-like view of the elements and values. Select the “Source” tab at the bottom of the editor pane to edit the XML source.

*web.xml* is an XML file with a root element of `<web-app>`. To map URL patterns to servlets, you declare each servlet with a `<servlet>` element, then declare the mapping with a `<servlet-mapping>` element. The `<url-pattern>` of a servlet mapping can be a full URL path, or a URL path with a `*` at the beginning or end to represent a part of a path. In this case, the URL pattern `/*` matches all URLs.

Be sure that each of your `<url-pattern>` values starts with a forward slash (`/`). Omitting the starting slash may have the intended behaviour on the development web server but unintended behavior on App Engine.

App Engine needs one additional configuration file that isn't part of the servlet standard. Open or create the file `war/WEB-INF/appengine-web.xml`, and give it contents similar to Example 3.134.

**Example 3.134. The `appengine-web.xml` file, with App Engine-specific configuration for the Java app**

```
<?xml version="1.0" encoding="utf-8"?>
<appengine-web-app xmlns="http://appengine.google.com/ns/1.0">
  <application>clock</application>
  <version>1</version>
</appengine-web-app>
```

In this example, the configuration file tells App Engine that this is version 1 of an application called *clock*. You can also use this configuration file to control other behaviors, such as static files and sessions.

The WAR for the application must include several JARs from the App Engine SDK: the Java EE implementation JARs, and the App Engine API JAR. The Eclipse plug-in installs these JARs in the WAR automatically. If you are not using the Eclipse plug-in, you must copy these JARs manually. Look in the SDK directory in the `lib/user/` and `lib/shared/` subdirectories. Copy every `.jar` file from these directories to the `war/WEB-INF/lib/` directory in your project.

Finally, the servlet class must be compiled. Eclipse compiles all of your classes automatically, as needed. If you are not using Eclipse, you probably want to use a build tool such as Apache Ant to compile source code and perform other build tasks. See the official App Engine documentation for information on using Apache Ant to build App Engine projects.

I suppose it's traditional to explain how to compile a Java project from the command line using the `javac` command. You can do so by putting each of the JARs from `war/WEB-INF/lib/` and the `war/WEB-INF/classes/` directory in the classpath, and making sure the compiled classes end up in the `classes/` directory. But in the real world, you want your IDE or an Ant script to take care of this for you. Also, when we introduce the datastore in the next few sections, we will need to add a step to building the project that makes this even more impractical to do by hand.

It's time to test this application with the development web server. The Eclipse plug-in can run the application and the development server inside the Eclipse debugger. To start it, select the Run menu, Debug As, and Web Application. The server starts, and prints the following message to the Console panel:

*The server is running at <http://localhost:8080/>*

If you are not using Eclipse, you can start the development server using the `dev_appserver` command (`dev_appserver.sh` for Mac OS X or Linux). The command takes the path to the WAR directory as an argument, like so:

*dev\_appserver war*

Test your application by visiting the server's URL in a web browser: <http://localhost:8080>.

### **The Development Console**

The Python and Java development web servers include a handy feature for inspecting and debugging your application while testing on your local machine: a web-based development console. With your development server running, visit the following URL in a browser to access the console:

*[http://localhost:8080/\\_ah/admin](http://localhost:8080/_ah/admin)*

You can also click the SDK Console button to open the console in a browser window. Figure 3.83 shows the datastore viewer in console.

The screenshot shows the 'clock Development Console' interface. On the left, a sidebar lists links: Datastore Viewer (selected), Interactive Console, Memcache Viewer, Task Queues, Cron Jobs, and XMPP. The main area is titled 'Datastore Viewer' and shows a table of entities for 'UserPrefs'. The table has columns: Key, ID, Key Name, tz\_offset, and user. One entity is listed: agVjbG9j... (ID 185804764220139124118), with tz\_offset -8 and user test@example.com. There are 'Delete' and '1' buttons at the bottom. The top right shows 'Results 1 - 1 of 1'.

Figure 3.83. The development console's datastore viewer

The Java development server also has a console. It includes a datastore viewer that lets you list and inspect datastore entities by kind, the ability to run task queues, and the ability to send email and XMPP messages to the app.

## Registering the Application

Before you can upload your application to App Engine and share it with the world, you must first create a developer account, then register an application ID. If you intend to use a custom domain name (instead of the free `appspot.com` domain name included with every app), you must also set up the Google Apps service for the domain. You can do all of this from the App Engine Administration Console.

To access the Administration Console, visit the following URL in your browser:

`https://appengine.google.com/`

Sign in using the Google account you intend to use as your developer account. If you don't already have a Google account (such as a Gmail account), you can create one using any email address.

Once you have signed in, the Console displays a list of applications you have created, if any, and a button to "Create an Application," similar to Figure 3.84. From this screen, you can create and manage multiple applications, each with its own URL, configuration, and resource limits.

The screenshot shows the 'My Applications' section of the Admin Console. It lists one application: 'clock' (Current Version 1). Below the table is a 'Create an Application' button and a note: 'You have 9 applications remaining.' The top right shows the user's email (juliet@example.com) and links for My Account, Help, and Sign out.

Figure 3.84. The Administration Console application list, with one app

When you register your first application ID, the Administration Console prompts you to verify your developer account using an SMS message sent to your mobile phone. After you enter your mobile phone number, Google sends an SMS to your phone with a confirmation code. Enter this code to continue the registration process. You

can verify only one account per phone number, so if you have only one mobile number (like most people), be sure to use it with the account you intend to use with App Engine.

If you don't have a mobile phone number, you can apply to Google for manual verification by filling out a web form. This process takes about a week. For information on applying for manual verification, see the official App Engine website.

You can have up to 10 active application IDs created by a given developer account. If you decide you do not want an app ID, you can disable it using the Administration Console to reclaim one of your 10 available apps. Disabling an app makes the app inaccessible by the public, and disables portions of the Console for the app. Disabling an app does *not* free the application ID for someone else to register. You can request that a disabled app be deleted permanently.

To disable or request deletion of an app, go to "Application Settings" in the Administration Console, and click the Disable Application... button. When you request deletion, all developers of the app are notified by email, and if nobody cancels the request, the app is deleted after 24 hours.

## The Application ID and Title

When you click the "Create an Application" button, the Console prompts for an application identifier. The application ID must be unique across all App Engine applications, just like an account username.

The application ID identifies your application when you interact with App Engine using the developer tools. The tools get the application ID from the application configuration file. For Python applications, you specify the app ID in the *app.yaml* file, on the *application:* line. For Java applications, you enter it in the *<application>* element of the *appengine-web.xml* file.

In the example earlier in this chapter, we chose the application ID "clock" arbitrarily. If you'd like to try uploading this application to App Engine, remember to edit the appropriate configuration file after you register the application to change the application ID to the one you chose.

The application ID is part of the domain name you can use to test the application running on App Engine. Every application gets a free domain name that looks like this:

*app-id.appspot.com*

The application ID is also part of the email and XMPP addresses the app can use to receive incoming messages.

Because the application ID is used in the domain name, an ID can contain only lowercase letters, numbers, or hyphens, and must be shorter than 32 characters. Additionally, Google reserves every Gmail username as an application ID that only the corresponding Gmail user can register. As with usernames on most popular websites, a user-friendly application ID may be hard to come by.

When you register a new application, the Console also prompts for an "application title." This title is used to represent your application throughout the Console and the rest of the system. In particular, it is displayed to a user when the application directs the user to sign in with a Google account. Make sure the title is as you would want your users to see it.

Once you have registered an application, its ID cannot be changed, though you can delete the application and create a new one. You can change the title for an app at any time from the Administration Console.

If you or other members of your organization want to use Google Apps accounts as developer accounts, you must access the Administration Console using a special URL. For example, if your Apps domain is *example.com*, you would use the following URL to access the Administration Console:

*https://appengine.google.com/a/example.com*

You sign in to the domain's Console with your Apps account.

If you create an app using a non-Apps account, then restrict its authentication to the domain, you will still be able to access the Administration Console using the non-Apps account. However, you will not be able to sign in to the app with that account, including when accessing URLs restricted to administrators.

## Uploading the Application

In a traditional web application environment, releasing an application to the world can be a laborious process. Getting the latest software and configuration to multiple web servers and backend services in the right order and at the right time to minimize downtime and prevent breakage is often difficult and delicate. With App Engine, deployment is as simple as uploading the files with a single click or command. You can upload and test multiple versions of your application, and set any uploaded version to be the current public version.

For Java apps, you can upload from Eclipse using the Google plug-in, or from a command prompt. In Eclipse, click the “Deploy to App Engine” button (the little App Engine logo) in the Eclipse toolbar. Or from a command prompt, run the *appcfg* (or *appcfg.sh*) command from the SDK’s *bin/* directory as follows, using the path to your application’s WAR directory for *war*:

```
appcfg update war
```

When prompted by these tools, enter your developer account’s email address and password. The tools remember your credentials for subsequent runs so you don’t have to enter them every time.

The upload process determines the application ID and version number from the app configuration file—*appengine-web.xml* for Java apps—then uploads and installs the files and configuration as the given version of the app. After you upload an application for the first time, you can access the application immediately using either the *.appspot.com* subdomain or the custom Google Apps domain you set up earlier. For example, if the application ID is *clock*, you can access the application with the following URL:

*http://clock.appspot.com/*

There is no way to download an application’s files from App Engine after they have been uploaded. Make sure you are retaining copies of your application files, such as with a revision control system and regular backups.

## Administration Console

You manage your live application from your browser using the App Engine Administration Console. You saw the Console when you registered the application, but as a reminder, you can access the Console at the following URL:

*https://appengine.google.com/*

If your app uses a Google Apps domain name and you are using an Apps account on the domain as your developer account, you must use the Apps address of the Administration Console:

*https://appengine.google.com/a/example.com*

Select your application (click its ID) to go to the Console for the app.

The first screen you see is the “dashboard”. The dashboard summarizes the current and past status of your application, including traffic and load, resource usage, and error rates. You can view charts for the request rate, the amount of time spent on each request, error rates, bandwidth and CPU usage, and whether your application is hitting its resource limits.

## References

1. CA Binildas, Malhar Barai, Vincenzo Caselli, *Service Oriented Architecture with Java. Using SOA and web services to build powerful Java applications*, Packt Publishing Ltd., 2008
2. Kenneth L. Calvert, Michael J. Donahoo. *TCP/IP sockets in Java : practical guide for programmers*, 2nd ed., Elsevier, 2008
3. Joe Clabby, *Web Services Explained: Solutions and Applications for the Real World*, Prentice Hall PTR, 2002
4. Naci Dai, Lawrence Mandel, Arthur Ryman, *Eclipse Web tools platform developing Java Web applications*, Pearson Education, Inc., 2007
5. Thomas Erl, *Service-Oriented Architecture: Concepts, Technology, and Design*, Prentice Hall PTR, 2005
6. Jim Farley, *Java Distributed Computing*, O'Reilly, 1998
7. Robert Flennner, Michael Abbott, Toufic Boubez, Frank Cohen, Navaneeth Krishnan, Alan Moffet, Jan Graba, *An Introduction to Network Programming with Java*, Springer, 2007
8. William von Hagen, *Professional Xen Virtualization*, Wiley Publishing, Inc., 2008
9. Brian Hochgurtel, *Cross-Platform Web Services Using C# and Java*, Charles River Media, 2003
10. Anthony T. Holdener, *Ajax: The Definitive Guide*, O'Reilly, 2008
11. John Paul Mueller, *Mining Google Web Services: Building Applications with the Google API*, Sybex, 2004
12. James Murty, *Programming Amazon Web Services*, O'Reilly, 2008
13. Ramesh Nagappan, Robert Skoczylas, Rima Patel Sriganesh, *Developing Java Web Services: Architecting and Developing Secure Web Services Using Java*, Wiley Publishing Inc., 2003
14. Eric Newcomer, Greg Lomow, *Understanding SOA with Web Services*, Addison Wesley Professional, 2004
15. Arno Puder, Kay Römer, Frank Pilhofer, *Distributed systems architecture: a middleware approach*, Elsevier, 2006
16. Rajam Ramamurti, Bilal Siddiqui, Frank Sommers, *Java P2P Unleashed*, Sams Publishing, 2002
17. Dan Sanderson, *Programming Google App Engine*, O'Reilly Media, 2012
18. Inderjeet Singh, Sean Brydon, Greg Murray, Vijay Ramachandran, Thierry Violleau, Beth Stearns, *Designing Web Services with the J2EE 1.4 Platform JAX-RPC, SOAP, and XML Technologies*, Addison Wesley, 2004
19. Andrew S. Tanenbaum, *Distributed Operating Systems*, Prentice Hall, 1994
20. Toby Velte, Antony Velte, Robert Elsenpeter, *Cloud Computing: A Practical Approach*, McGraw-Hill, 2009