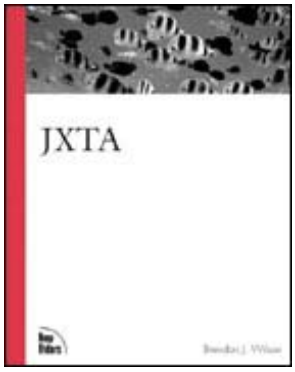


Front Matter
Table of Contents
About the Author

JXTA

Brendon J. Wilson

Publisher: New Riders Publishing
First Edition June 11, 2002
ISBN: 0-73571-234-4, 512 pages

Explore the hottest new paradigm, peer-to-peer (P2P) computing, and use the power of the JXTA platform to transform your applications into peer-aware, collaborative tools. JXTA technology is a set of open source, peer-to-peer protocols that allow any connected device on the network from cell phone to PDA from PC to server to communicate and collaborate in a peer to peer manner. With the explosion of content on the network, and the existence of millions of connected devices, a multi-dimensional web or expanded web has emerged. Content is both on the edge of the network, but also resides in the "deep web." JXTA technology enables new and innovative network applications to be created, giving complete access to content on the expanded web. This book is an implementation book that covers the protocols and how to use them.

Table of Content

About the Author

Brendon J. Wilson, a graduate of Simon Fraser University's Engineering Science program (www.ensc.sfu.ca), is a software engineer specializing in object-oriented programming with a focus on Java and Internet technologies. Brendon started using Java in 1996 as part of his undergraduate thesis project, a 3D robot manipulator simulator, which went on to win Sun Microsystems' Java3D programming competition.

Since graduating from SFU, Brendon has worked at a number of high-tech software-development companies, including the e-business division of IBM's Pacific Development Center in Burnaby, and a variety of encryption and wireless startups around the world. Currently a Senior Software Engineer at PKI Innovations, Inc. (www.pk3i.com), Brendon divides his time between his job, his pursuit of his PEng (professional engineer) designation, and investigations into all the latest Internet technologies. Occasionally he sleeps, too.

Brendon lives with his wife, Ashley, in Vancouver, Canada, where he is currently recovering from an all-consuming addiction to *The Simpsons*. He can be contacted through his web site at www.brendonwilson.com.

About the Technical Reviewers

These reviewers contributed their considerable hands-on expertise to the entire development process for *JXTA*. As the book was being written, these dedicated professionals reviewed all the material for technical content, organization, and flow. Their feedback was critical to ensuring that *JXTA* fits our readers' need for the highest-quality technical information.

William R. Bauer retired from the defense industry in 1997 to help his talented and gorgeous wife raise their newborn son. He currently has a small consulting practice catering primarily to medical research. He had a diverse 30-year career in defense as a software, hardware, and algorithm designer. His techniques for measuring Dingle temperatures in solid-state physics and for realtime measurement of pitch in voice processing are still in use.

Bill (a.k.a.Vasha) joined the JXTA project shortly after it became open-source. He co-owns the JXTA-Wire and JXTA-httpd projects, and has learned a great deal from the originator of those projects, Eric Pouyoul. Presently Bill is developing transports and services optimized for JXTA's Voice Over P2P (vop2p) project.

Chris Genly has more than 25 years of software engineering experience in diverse subjects such as natural language processing, compilers, and distributed computing. Current interests include Java, P2P, and eXtreme Programming. Chris lives in the idyllic town of Forest Grove, Oregon, with his wife and two children.

Acknowledgments

First, I'd like to thank Sun Microsystems and the Project JXTA development team for creating the JXTA technology and releasing it to the JXTA Community.

Second, I'd like to thank all the members of the JXTA Community and JXTA mailing lists for providing me with invaluable constructive criticism on early drafts of the book. This feedback allowed me to tune the book and catch numerous errors early in its development. More important, the community provided me with encouragement when I needed it most.

Third, I'd like to thank all of those involved in the production of this book. In particular, I'd like to recognize Jeff Riley (I owe my life to him), Stephanie Wall, Rubi Olis, Elise Walter, and Lisa Thibault at New Riders. I'd also like to thank New Riders for not only giving me the opportunity to write this book, but also having the foresight to allow me to release early drafts to the JXTA Community for review. At a time when many companies are greedily hoarding intellectual property, it is encouraging to see that a company like New Riders can recognize the benefit that it can realize from contributing freely to an online community.

Fourth, I'd also like to thank my two phenomenal technical editors, William Bauer and Chris Genly, for their insightful comments and flattering remarks. Without them, this book wouldn't have been half as useful as it is. John Harvey deserves a mountain of credit for somehow making me look intelligent for the book's photo, despite losing several lenses in the process.

Fifth, I'd like to thank Mr. Paul Knipe, Mr. Mark Van Camp, Mr. Rod Osiowy, Dr. John Dill, Dr. Ash Parameswaran, and the many other teachers I've had throughout my secondary and post-secondary education. These are the people who encouraged me to make the most of myself and to try new things. The future is in good hands as long as we have dedicated teachers like these educating our children.

Finally, and most important, I'd like to thank my wife, Ashley; my parents, Mae and Rod; and all of my friends for their love and support—oh, and for putting up with me when I start ranting or babbling (my only two forms of "conversation").

Tell Us What You Think

As the reader of this book, you are the most important critic and commentator. We value your opinion and want to know what we're doing right, what we could do better, what areas you'd like to see us publish in, and any other words of wisdom you're willing to pass our way.

As the Associate Publisher for New Riders Publishing, I welcome your comments. You can fax, email, or write me directly to let me know what you did or didn't like about this book—as well as what we can do to make our books stronger.

Please note that I cannot help you with technical problems related to the topic of this book and that, due to the high volume of mail I receive, I might not be able to reply to every message.

When you write, please be sure to include this book's title and author as well as your name and phone or fax number. I will carefully review your comments and share them with the author and editors who worked on the book.

Fax:	317-581-4663
Email:	stephanie.wall@newriders.com
Mail:	Stephanie Wall Associate Publisher New Riders Publishing 201 West 103rd Street Indianapolis, IN 46290 USA

Preface

The explosion in the number of peer-to-peer (P2P) solutions has underlined the importance of this technology to the continued growth and development of the Internet. However, the proprietary and specialized nature of current solutions highlights the need for a standard set of protocols to address the particular requirements of the P2P domain. The JXTA platform provides developers with a flexible, standards-based set of protocols and reference libraries. Using JXTA, developers can focus on implementing their collaborative applications rather than on the specifics of P2P technology.

The JXTA platform is the next stage of development in the maturing P2P arena, a fundamental shift in the way people use the Internet. Using JXTA, developers can create new applications that will communicate with any number of distributed peers, providing distributed search, file sharing, and collaboration services without relying on the traditional client/server hierarchy. The result is more robust and reliable applications that enable users to fully realize the communication capabilities of the Internet.

What This Book Covers

This book presents a guided tour of the JXTA platform, including all the critical information required to begin producing P2P solutions built on top of JXTA. Reference information on each of the JXTA protocols provides an understanding of the underlying principles of P2P networking, and examples built on the JXTA reference implementation provide the hands-on experience necessary to become fluent in JXTA technology.

All examples in the book use the JXTA reference implementation written in Java™, allowing the example code to be run on any device with a Java Virtual Machine (JVM), thereby reaching the largest possible development audience. For developers who want to use another development language, the protocol reference information within the book is comprehensive enough to allow an advanced developer to produce solutions that are compatible with JXTA.

Chapter 1, “Introduction,” is a general introduction to the topic of peer-to-peer (P2P) computing, including the history of the development of the P2P paradigm, the advantages and disadvantages of P2P, and an introduction to Project JXTA.

Chapter 2, "P2P Concepts," introduces the basic concepts of P2P networking, the problems posed by P2P, and the terminology required to understand the rest of the book. This chapter provides a technical introduction to the components required to implement a complete P2P network.

Chapter 3, "Introducing JXTA P2P Solutions," introduces JXTA and the solutions that it provides to address basic problems in P2P networking. It discusses the design philosophy, assumptions, capabilities, and limitations of JXTA. The majority of the chapter is devoted to using the JXTA Shell application, to familiarize the reader with the JXTA implementation of P2P, and to allow some preliminary experimentation without requiring the reader to program anything just yet. In addition, a small introduction on XML familiarizes the reader with the form of messages used by all the JXTA protocols.

Chapter 4, "The Peer Discovery Protocol," details the Peer Discovery Protocol (PDP), which provides JXTA P2P applications with a mechanism for discovering other peer resources. Without the PDP, a P2P client would be useless, incapable of finding and using the resources offered by other peers. The chapter elaborates on the purpose of the PDP, its use in JXTA applications, and the format of PDP messages. Examples, using the JXTA Shell and Java code written using the reference JXTA implementation, guide the reader through the use of the PDP to discover other peer resources on the network.

Chapter 5, "The Peer Resolver Protocol," discusses the Peer Resolver Protocol (PRP), which provides P2P applications with a generic request-and-response format to use when communicating with other peers. After a peer has been discovered using the Peer Discovery Protocol, the PRP can be used to send messages to the peer for processing and to receive messages from the peer containing the results. This chapter details the purpose of the PRP, the use of the PRP in JXTA applications, and the format of PRP messages; it also guides the reader through example code that uses the PRP to send and receive simple messages between two peers.

Chapter 6, "The Rendezvous Protocol," details the Rendezvous Protocol (RVP), used by a peer to connect to a rendezvous peer and have messages propagated on its behalf to other peers that are also connected to the rendezvous peer. Rendezvous peers provide a mechanism for peers to broadcast messages to many peers without relying on a specific network transport. This chapter provides information on the format of the RVP messages and the flow of messages between a peer and the rendezvous peer. This chapter also covers

the Rendezvous service's dual role, providing both a local interface to remote rendezvous peers and rendezvous peer services to remote peers.

Chapter 7, "The Peer Information Protocol," discusses the Peer Information Protocol (PIP). After a peer has been discovered using the Peer Discovery Protocol, the status or capabilities of the peer might be required. The PIP provides a set of messages capable of querying a peer to obtain status information. This chapter details the purpose of the PIP, the use of the PIP in JXTA applications, and the format of PIP messages; it also guides the reader through an example application that uses the PIP to send and receive messages to communicate peer status information.

Chapter 8, "The Pipe Binding Protocol," covers the Pipe Binding Protocol (PBP). Pipes in JXTA provide a virtual communication channel connecting endpoints on the P2P network. The PBP allows peer group members to establish a connection to another peer, independent of the transport mechanism. This chapter details the purpose of the PBP, the use of the PBP in P2P applications, and the format of PBP messages; it also guides the reader through an example application that uses the PBP to exchange messages over a pipe with another peer.

Chapter 9, "The Endpoint Routing Protocol," discusses the Endpoint Routing Protocol (ERP). Due to the ad hoc nature of a P2P network, a mechanism is required to enable messages to be routed between peers. The ERP provides peers with a mechanism for determining a route to an endpoint. This routing mechanism is provided transparently by the Endpoint service, allowing a peer to send messages without needing to take special steps to handle communication via intermediary peers. This chapter details the purpose of the ERP and the format of the ERP's messages, and it guides the user through an example application that uses the ERP and the Endpoint service to send messages to another peer endpoint.

Chapter 10, "Peer Groups and Services," covers the use of peer groups to segment the network space and provide new services on the JXTA network. The chapter discusses peer group creation and configuration, as well the process of bootstrapping the JXTA platform. Much coverage of the topic of modules is provided, and the chapter's example demonstrates the creation of a new peer group service and the creation of a peer group configured to use the new peer group service.

Chapter 11, "A Complete Sample Application," guides the reader through the process of creating a complete P2P solution using all the JXTA protocols. Examples from each of the preceding chapters are used as the foundation of the application and are brought together to provide all the elements of the final P2P solution.

Chapter 12, "The Future of JXTA," outlines some of the future directions currently being pursued by JXTA project groups, including implementations of JXTA for other languages and bindings to other network transports. In addition, this chapter introduces some of the other community projects that build on JXTA to provide services and applications.

Two appendixes provide a glossary of terms and three-letter acronyms (TLA) used in the book, as well as a list of related online resources.

Who Is this Book For?

This book is targeted at software developers doing peer-to-peer application development who are interested in detailed information on the JXTA platform technologies and concepts. This book assumes an intermediate level of Java development knowledge and a basic knowledge of networking. Developers who are not familiar with Java should still be able to understand and run the book's example code with a minimum of difficulty.

Conventions Used in this Book

This book follows a few typographical conventions:

- A new term is set in *italics* the first time it is introduced.
- Program text, functions, variables, and other "computer language" are set in a fixed-pitch font—for example, `<Person>`.
- When a line of code wraps to a new line, a code continuation character (➥) is used to indicate.

When there's additional information to the discussion, I'll add a sidebar that looks like this:

The Tragedy of the Commons

In many communities that share resources, there is a risk of suffering from the Tragedy of the Commons: the overuse of a shared resource to the point of its destruction. The Tragedy of the Commons originally referred to the problem of overgrazing on public lands, but the term can apply to any public resource that can be used without restriction.

When there's a tip that I want to share, I'll add a note that looks like this:

Note

This sets a system property called `net.jxta.tls.password` to the `password` value provided after the equals (=) sign and sets a system property called `net.jxta.tls.principal` to the `username` provided. When you start the Shell from the command line and include these parameters, the Shell starts immediately without prompting for your username and password.

What You'll Need to Try the Examples

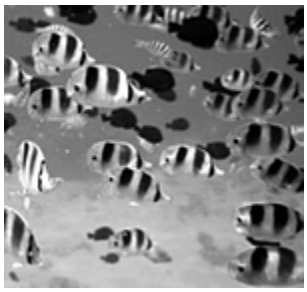
The examples and screenshots in this book were created using the Java 2 SDK Standard Edition version 1.3.1 from Sun Microsystems running on Windows 2000. Although Java runs on a variety of operating systems, it's still safe to say that most people are running Windows. However, all the example applications should run on any operating system with an implementation of the Java 2 SDK and JVM version 1.3.1 or later. If you are running the examples on a non-Windows system, you might need to translate some commands from Windows to your own operating system's equivalent commands.

To download the Java 2 SDK Standard Edition for Windows 2000 or a number of other platforms, go to www.javasoft.com/j2se/ and download the appropriate SDK for your operating system. For Mac users, a Mac implementation of the Java 2 Standard Edition SDK is available from www.apple.com/java/, but only for users of the Mac OS X operating system.

All the Java source code for the examples discussed in this book is available for download from the New Riders web site at www.newriders.com.

Part I: The Basics of JXTA

Chapter 1. Introduction

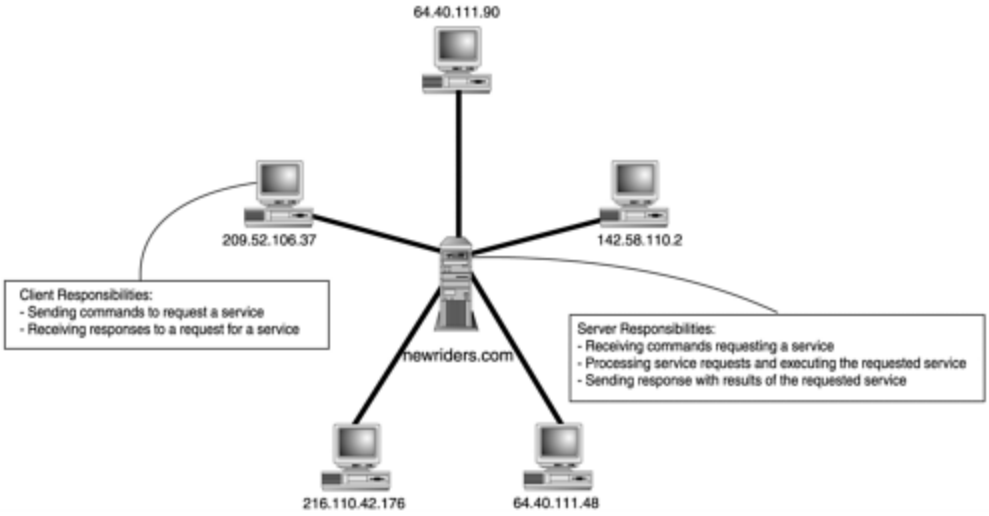


Peer-to-peer (P2P) technology enables any network-aware device to provide services to another network-aware device. A device in a P2P network can provide access to any type of resource that it has at its disposal, whether documents, storage capacity, computing power, or even its own human operator. Although P2P might sound like a dot-com fad, the technology is a natural extension of the Internet's philosophy of robustness through decentralization. In the same manner that the Internet provides domain name lookup (DNS), World Wide Web, email, and other services by spreading responsibility among millions of servers, P2P has the capacity to power a whole new set of robust applications by leveraging resources spread across all corners of the Internet.

Introduction to Peer-to-Peer

Most Internet services are distributed using the traditional client/server architecture, illustrated in Figure 1.1. In this architecture, clients connect to a server using a specific communications protocol, such as the File Transfer Protocol (FTP), to obtain access to a specific resource. Most of the processing involved in delivering a service usually occurs on the server, leaving the client relatively unburdened. Most popular Internet applications, including the World Wide Web, FTP, telnet, and email, use this service-delivery model.

Figure 1.1. Client/server architecture.



Unfortunately, this architecture has a major drawback. As the number of clients increases, the load and bandwidth demands on the server also increase, eventually preventing the server from handling additional clients. The advantage of this architecture is that it requires less computational power on the client side. Ironically, most users have been persuaded to upgrade their computer systems to levels that are ludicrously overpowered for the most popular Internet applications: surfing the web and retrieving email.

The client in the client/server architecture acts in a passive role, capable of demanding services from servers but incapable of providing services to other clients. This model of service delivery was developed at a time when most machines on the Internet had a resolvable static IP address, meaning that all machines on the Internet could find each other easily using a simple name (such as `yourmachine.com`). If all machines on the network ran both a server and a client, they formed the foundation of a rudimentary P2P network.

As the Internet grew, the finite supply of IP addresses prompted service providers to begin dynamically allocating IP addresses to machines each time they connected to the network through dial-up connections. The dynamic nature of these machines' IP addresses effectively prevented users from running useful servers. Although someone could still run a server, that user couldn't access it unless he knew the machine's IP address beforehand. These computers form the "edge" of the Internet: machines that are connected but incapable of easily participating in the exchange of services. For this reason, most useful

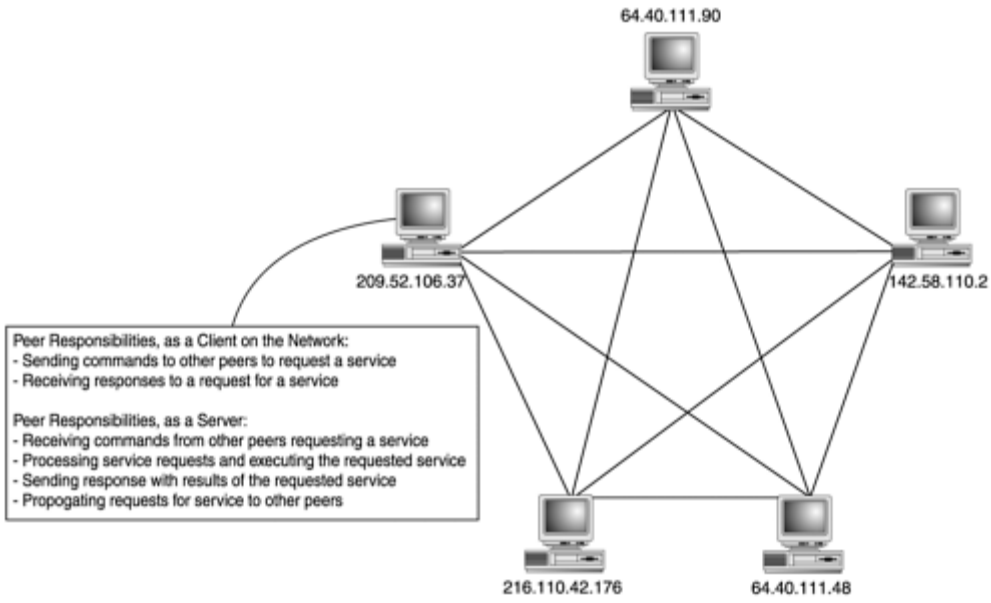
services are centralized on servers with resolvable IP addresses, where they can be reached by anyone who knows the server's easy-to-remember domain name.

Another reason that most clients' machines can't run servers is that they are a part of a private network, usually run by their own corporation's IT department. This private network is usually isolated from the Internet by a firewall, a device designed to prevent arbitrary connections into and out of the private network. Corporations usually create a private network to secure sensitive corporate information as well as to prevent against network abuse or misuse. The side effect of this technology is that a computer outside the private network can't connect to a computer within the private network to obtain services.

Consider the amount of computing and storage power that these client machines represent! Assume that only 10 million 100MHz machines are connected to the Internet at any one time, each possessing only 100MB of unused storage space, 1000bps of unused bandwidth, and 10% unused processing power. At any one time, these clients represent 10 petabytes (PB) (10^{15} bytes) of available storage space, 10 billion bps of available bandwidth (approximately 1.25GBps), and 10^8 MHz of wasted processing power! These are conservative estimates that only hint at the enormous untapped potential waiting to be unleashed from the "edge" of the Internet.

P2P is the key to realizing this potential, giving individual machines a mechanism for providing services to each other. Unlike the client/server architecture, P2P networks don't rely on a centralized server to provide access to services, and they usually operate outside the domain name system. As shown in Figure 1.2, P2P networks shun the centralized organization of the client/server architecture and instead employ a flat, highly interconnected architecture. By allowing intermittently connected computers to find each other, P2P enables these machines to act as both clients and servers that can determine the services available on the P2P network and engage those services in some application-specific manner.

Figure 1.2. Peer-to-peer architecture.



The main advantage of P2P networks is that they distribute the responsibility of providing services among all peers on the network; this eliminates service outages due to a single point of failure and provides a more scalable solution for offering services. In addition, P2P networks exploit available bandwidth across the entire network by using a variety of communication channels and by filling bandwidth to the "edge" of the Internet. Unlike traditional client/server communications, in which specific routes to popular destinations can become overtaxed (for example, the route to Amazon.com), P2P enables communication via a variety of network routes, thereby reducing network congestion.

P2P has the capability of serving resources with high availability at a much lower cost while maximizing the use of resources from every peer connected to the P2P network. Whereas client/server solutions rely on the addition of costly bandwidth, equipment, and co-location facilities to maintain a robust solution, P2P can offer a similar level of robustness by spreading network and resource demands across the P2P network. Companies such as Intel are already using P2P to reduce the cost of distributing documents and files across the entire company.

Unfortunately, P2P suffers from some disadvantages due to the redundant nature of a P2P network's structure. The distributed form of communications channels in P2P networks results in service requests that are nondeterministic in nature. For example, clients

requesting the exact same resource from the P2P network might connect to entirely different machines via different communication routes, with different results. Requests sent via a P2P network might not result in an immediate response and, in some cases, might not result in *any* response. Resources on a P2P network can disappear at times as the clients that host those resources disconnect from the network; this is different from the services provided by the traditional Internet, which have most resources continuously available.

However, P2P can overcome all these limitations. Although resources might disappear at times, a P2P application might implement functionality to mirror the most popular resources over multiple peers, thereby providing redundant access to a resource. Greater numbers of interconnected peers reduce the likelihood that a request for a service will go unanswered. In short, the very structure of a P2P network that causes problems can be used to solve them.

Why Is Peer-to-Peer Important?

Although P2P gained notoriety as a means for illegally distributing copyrighted intellectual property, P2P has more to offer the computing world than easy access to stolen music or video files. To illustrate the difference between the way things are done now and how P2P could provide more useful and robust solutions, consider the following example.

To find some specific information on the Internet, I usually point my web browser to my favorite search engine, Google, and submit a search query. Most times, I'll receive a list of several thousand results, many of which are unrelated, are out-of-date, or worse yet, point to resources that no longer exist. How frustrating!

One of the problems with the current search engine solution lies in the centralization of knowledge and resources. Google relies on a central database that is updated daily by scouring the Internet for new information. Due to the number of indexed web pages in its database (more than 1.6 billion), not every entry gets updated every day. As a result of this shortcoming, the information in the Google database might not reflect the most up-to-date information available, thus diminishing the usefulness of its results for any given search query.

The search engine technology has a number of other disadvantages:

- It requires a lot of equipment. Google, for example, runs a Linux cluster of 10,000 machines to provide its service.

- If the search engine goes offline (due to, say, a network outage), all the search engine's information is unavailable.

- Due to the size of the Internet, the search engine cannot provide a comprehensive index of the Internet.

- Search engines can't interface with information stored in a corporate web site's database, meaning that the search engine can't "see" some information.

A similar service could be implemented using P2P technology, augmenting the service with additional desirable properties. Imagine if every person could run a personal web server on a desktop computer! Suppose that, in addition to serving content from the user's machine, this server had the capability to process requests for information about the documents managed by the server. A user's server could receive a query, check the documents that it manages for a match, and respond to the query with a list of matching documents.

The user's server would be responsible for indexing the documents that it made available and therefore would be capable of providing more accurate, up-to-date information on the user's documents to anyone submitting a search query. The task of indexing a single user's documents would be much more manageable than the task facing Google (a couple dozen web pages versus billions of pages). Corporations could provide gateways to connect their own web sites' databases of information to the P2P network, providing searchable access to information that the search engines currently can't reach.

The system would have this added advantage: If the user's server disconnected from the network, the search service would also become unavailable; users searching the network wouldn't receive results for resources that were unavailable. As someone searching for information, I would be almost guaranteed that any result I found using the system would be available, reducing wasted search time. I could even sort search results from the entire network to determine which information might suit my needs better based on various characteristics (such as the responsiveness of the server hosting a resource or the number of servers hosting a copy of the same resource).

This example application of P2P technology isn't perfect. For one thing, anyone wanting to drive traffic to a site could return that site as a match to any search query. However, the example illustrates the underlying principle of P2P: to enable anyone to offer services over a network. Until now, the traditional Internet experience has been mostly passive. Like the desktop publishing revolution of the mid-1980s, P2P promises to revolutionize the exchange of information.

A Brief History of P2P

Peer-to-peer has always existed, but it hasn't always been recognized as such; servers with fixed or resolvable IP addresses have always had the capability to communicate with other servers to access services. A number of pre-P2P applications, such as email and the domain name system, built on these capabilities to provide distributed networks, but one such application, Usenet, stands out from the others.

Usenet was created in 1979 by two North Carolina grad students, Tom Truscott and Jim Ellis, to provide a way for two computers to exchange information in the early days before ubiquitous Internet connectivity. Their first iteration allowed a computer to dial another computer, check for new files, and download those files; this was done at night to save on long-distance telephone charges. The system evolved into the massive newsgroup system that it is today. However, as large as Usenet is, it has a few properties that help distinguish it as probably the first P2P application. Usenet has no central managing authority—the distribution of content is managed by each node, and the content of the Usenet network is replicated (in whole or in part) across its nodes.

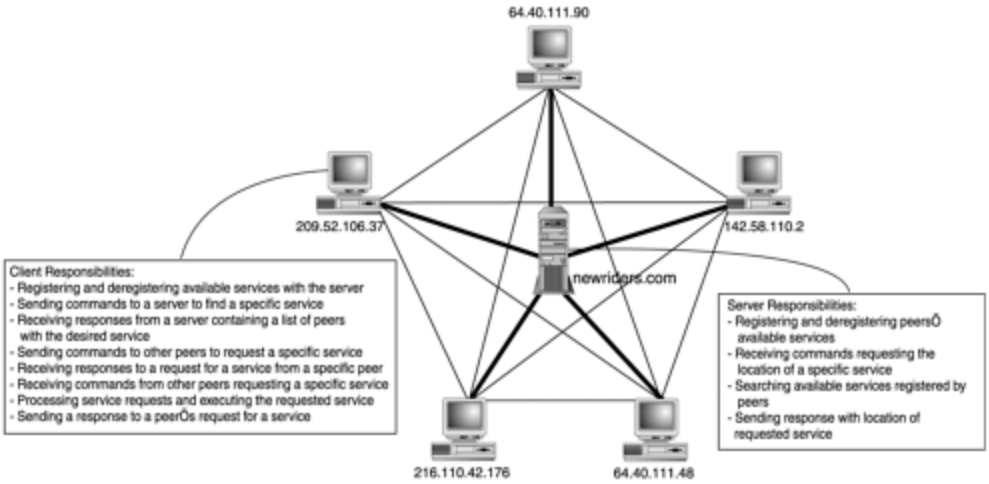
One of the most interesting things about Usenet is what it is: nothing! Usenet isn't a piece of software or a network of servers; although it requires software and servers to operate, these things don't truly define Usenet. At its core, Usenet is simply a way for machines to talk to each other to allow news messages to be posted and disseminated over a network. By providing a well-defined protocol, the Network News Transport Protocol (Internet Engineering Task Force RFC 977), the widest possible number of machines can participate independently to provide services. This distribution of responsibility is what distinguishes Usenet, making it recognizable as the first true, though rudimentary, application of P2P technology.

Since Usenet, the most popular P2P applications have fallen into one of three major categories: instant messaging, file sharing, and distributed computing.

Instant Messaging (IM)

When Mirabilis released ICQ (www.icq.com) in November 1996, it gave its users a faster way to communicate with friends than traditional email. ICQ allows users to be notified when their friends come online and to send instant messages to their friends. In addition to its main capability of instant messaging, ICQ allows users to exchange files. Though classified as a P2P application, ICQ relies on a hybrid of the P2P and client/server architectures to provide its service, as shown in Figure 1.3. ICQ uses a central server to monitor which users are currently online and to notify interested parties when new users connect to the network. All other communication between users is conducted in a P2P fashion, with messages flowing directly from one user's machine to another's with no server intermediary.

Figure 1.3. Hybrid P2P architecture.



Since its unveiling, ICQ has had many imitators, including MSN Messenger (www.messenger.msn.com), AOL Internet Messenger (www.aol.com/aim), and Yahoo! Messenger (www.messenger.yahoo.com). Sadly, these applications are not compatible; each relies on its own proprietary communication protocol. As a result of this incompatibility, users must download different client software and go through a separate registration process for each network. Because most users choose to avoid this

inconvenience, these networks have grown into completely separate user communities that cannot interact.

More recently, various software developers have tried to bridge these separate communities by reverse-engineering the IM protocols and making new client software. One such application, Jabber (www.jabber.com), provides gateways to all major IM services, allowing users to interact with each other across the various IM networks. This attempt has met with resistance from service providers, prompting AOL to change its communication protocol in an attempt to block Jabber clients.

File Sharing

Napster (www.napster.com) burst onto the Internet stage in 1999, providing users with the capability to swap MP3 files. Napster employs a hybrid P2P solution similar to ICQ, relying on a central server to store a list of MP3 files on each user's machine. This server is also responsible for allowing users to search that list of available files to find a specific song file and its host. File transfer functionality is coordinated directly between peers without a server intermediary. In addition to its main file-sharing functionality, Napster provides a chat function to allow users to send text messages to each other.

Taking its cue from Napster, but noting the legal implications of enabling copyright infringement, the Gnutella project (www.gnutelliums.com) took the file-sharing concept pioneered by Napster one step further and eliminated the need for a central server to provide search functionality. The Gnutella network's server independence, combined with its capability to share any type of file, makes it one of the most powerful demonstrations of P2P technology.

Peers on the Gnutella network are responsible not only for serving files, but also for responding to queries and routing messages to other peers. Note that although Gnutella doesn't require a central server to provide search and IP address resolution functionality, connecting to the Gnutella network still requires that a peer know the IP address of a peer already connected to the P2P network. For this reason, a number of peers with static or resolvable IP addresses have been established to provide new peers with a starting point for discovering other peers on the network.

Eliminating the reliance on a central server has raised a number of new issues:

- How do peers distribute messages to each other without flooding the network?
- How do peers provide content securely and anonymously?
- How can the network encourage resource sharing?

Other file-sharing P2P variants, including Freenet (freenet.sourceforge.net), Morpheus (www.musiccity.com), and MojoNation (www.mojonation.net), have stepped into the arena to address these issues. Each of these applications addresses a specific issue. Freenet provides decentralized anonymous content storage protected by strong cryptography against tampering. Morpheus provides improved search capabilities based on metadata embedded in common media formats. MojoNation uses an artificial currency, called Mojo, to enforce resource sharing.

The Tragedy of the Commons

In many communities that share resources, there is a risk of suffering from the "Tragedy of the Commons": the overuse of a shared resource to the point of its destruction. The Tragedy of the Commons originally referred to the problem of overgrazing on public lands, but the term can apply to any public resource that can be used without restriction.

In some P2P systems, peers can use the resources (bandwidth and storage space) of others on the network without making resources of their own available to the network, thereby reducing the value of the network. As more users choose not to share their resources, those peers that do share resources come under increased load and, in many ways, the network begins to revert to the classic client/server architecture. Taken to its logical conclusion, the network eventually collapses, benefiting no one.

Newer P2P solutions have tried to prevent the Tragedy of the Commons by incorporating checks to ensure that users share resources. Lime Wire (www.limewire.com), for example, allows users to restrict downloads based on the number of files that a requesting client is sharing with the network. MojoNation (www.mojonation.net) takes this model one step further and incorporates a system of currency that users earn by sharing resources and then spend to access resources.

Distributed Computing

Distributed computing is a way of solving difficult problems by splitting the problem into subproblems that can be solved independently by a large number of computers. Although the most popular applications of distributed computing have not been P2P solutions, it is important to note the breakthrough work that has been accomplished by projects such as SETI@Home (setiathome.berkeley.edu) and Distributed.net (distributed.net) and companies such as United Devices (www.ud.com).

In 1996, SETI@Home began distributing a screen saver–based application to users, to allow them to process radio-telescope data and contribute to the search for extraterrestrial life. Since then, it has signed up more than 3 million users (of which more than a half million are active contributors). In a similar project started in 1997, Distributed.net used the computing power of its users to crack previously unbreakable encrypted messages. In both cases, the client software contacts a server to download its portion of the problem

being solved; until the problem is solved, no further communication with the server is required.

In the future, it is expected that distributed computing will evolve to take full advantage of P2P technology to create a marketplace for spare computing power.

Introducing Project JXTA

As you probably noticed, most of the P2P solutions overlap in some shape or form: ICQ provides instant messaging plus a bit of file sharing. Napster provides file sharing plus a bit of instant messaging.You could even say that Gnutella provides file sharing, plus a bit of distributed computing, due to the way that peers take on the task of routing messages across the network.

Regrettably, the current applications of P2P tend to use protocols that are proprietary and incompatible in nature, reducing the advantage offered by gathering devices into P2P networks. Each network forms a closed community, completely independent of the other networks and incapable of leveraging their services.

Until now, the excitement of exploring the possibilities of P2P technology has overshadowed the importance of interoperability and software reuse. To evolve P2P into a mature solution platform, developers need to refocus their efforts from programming P2P network fundamentals to creating P2P applications on a solid, well-defined base. To do this, P2P developers need a common language to allow peers to communicate and perform the fundamentals of P2P networking.

Realizing this need for a common P2P language, Sun Microsystems formed Project JXTA (pronounced *juxtapose* or *juxta*), a small development team under the guidance of Bill Joy and Mike Clary, to design a solution to serve all P2P applications. At its core, JXTA is simply a set of protocol specifications, which is what makes it so powerful. Anyone who wants to produce a new P2P application is spared the difficulty of properly designing protocols to handle the core functions of P2P communication.

What Does JXTA Mean?

The name JXTA is derived from the word *juxtapose,* meaning to place two entities side by side or in proximity. By choosing this name, the development team at Sun recognized that P2P solutions would always exist alongside the current client/server solutions rather than replacing them completely.

The JXTA v1.0 Protocols Specification defines the basic building blocks and protocols of P2P networking:

- **Peer Discovery Protocol—** Enables peers to discover peer services on the network
- **Peer Resolver Protocol—** Allows peers to send and process generic requests
- **Rendezvous Protocol—** Handles the details of propagating messages between peers
- **Peer Information Protocol—** Provides peers with a way to obtain status information from other peers on the network
- **Pipe Binding Protocol—** Provides a mechanism to bind a virtual communication channel to a peer endpoint
- **Endpoint Routing Protocol—** Provides a set of messages used to enable message routing from a source peer to a destination peer

The JXTA protocols are language-independent, defining a set of XML messages to coordinate some aspect of P2P networking. Although some developers in the P2P community protest the use of such a verbose language, the choice of XML allows implementers of the JXTA protocols to leverage existing toolsets for XML parsing and formatting. In addition, the simplicity of the JXTA protocols makes it possible to implement P2P solutions on any device with a "digital heartbeat," such as PDAs or cell phones, further expanding the number of potential peers.

In April 2001, Bill Joy placed Project JXTA in the hands of the P2P development community by adopting a license based on the Apache Software License Version 1.1. In addition to maintaining the JXTA v1.0 Protocols Specification, Project JXTA is

responsible for the development of reference implementations of the JXTA platform and source code control for a variety of JXTA Community projects. Currently, Project JXTA has a reference implementation available in Java, with implementations in C, Objective-C, Ruby, and Perl 5.0 under way. At this time, Project JXTA houses a variety of JXTA Community projects that are applying JXTA technology in diverse fields such as content management, artificial intelligence, and secure anonymous payment systems.

Summary

This chapter provided an introduction to P2P and outlined the problems of the traditional client/server architecture that P2P can be used to solve. The advantages and shortcomings of current P2P solutions were presented, and the JXTA solution was briefly introduced.

The next chapter examines the common problems that face P2P implementations and how they can be solved. These solutions are presented independently of the JXTA technology but use the JXTA terminology. This allows the chapter to provide a high-level overview of P2P that doesn't overwhelm the reader with JXTA-specific details.

Chapter 2. P2P Concepts

It's necessary to introduce the terminology and concepts of JXTA and place them in the general framework that's common to all P2P networks. This chapter introduces the terminology used to describe aspects of P2P networks, the components common to all P2P solutions (including those not built using JXTA technology), and the problems and solutions inherent in P2P networks.

Elements of P2P Networks

P2P is the solution to a straightforward question: How can you connect a set of devices in such a way that they can share information, resources, and services? On the surface, it seems a simple question, but to answer it properly requires answering several implied questions:

- How does one device learn of another device's presence?
- How do devices organize to address common interests?
- How does a device make its capabilities known?
- What information is required to uniquely identify a device?
- How do devices exchange data?

All P2P networks build on fundamental elements to provide the answers to these questions and others. Unfortunately, many of these elements are assumed or implied by proprietary P2P networks and are hard-coded into many P2P applications' implementations, resulting in inflexibility. For example, the majority of current P2P solutions assumes the use of TCP

as a network transport mechanism and cannot operate in any other network environment. Flexible P2P solutions need a language that explicitly declares all of the variables in any P2P solution.

The following sections define the basic terminology of P2P networking. I've tried to provide definitions at this point that use the JXTA terminology while omitting the JXTA-specific implementation details. This will help you learn the language of JXTA and P2P without being overwhelmed by JXTA-specific details.

Peers

A *peer* is a node on a P2P network that forms the fundamental processing unit of any P2P solution. Until now, you might have described a peer as an application running on a single computer connected to a network such as the Internet, but that limited definition wouldn't capture the true function of a peer and all its possible incarnations. This limited definition discounts the possibility that a peer might be an application distributed over several machines or that a peer might be a smaller device, such as a PDA, that connects to a network indirectly, such as via a synching cradle. A single machine might even be responsible for running multiple peer instances.

To encompass all these facets, this book defines a peer as follows:

Any entity capable of performing some useful work and communicating the results of that work to another entity over a network, either directly or indirectly.

The definition of *useful work* depends on the type of peer. Three possible types of peers exist in any P2P network:

- Simple peers
- Rendezvous peers
- Router peers

Each peer on the network can act as one or more types of peer, with each type defining a different set of responsibilities for the peer to the P2P network as a whole.

Simple Peers

A simple peer is designed to serve a single end user, allowing that user to provide services from his device and consuming services provided by other peers on the network. In all likelihood, a simple peer on a network will be located behind a firewall, separated from the network at large; peers outside the firewall will probably not be capable of directly communicating with the simple peer located inside the firewall.

Because of their limited network accessibility, simple peers have the least amount of responsibility in any P2P network. Unlike other peer types, they are not responsible for handling communication on behalf of other peers or serving third-party information for consumption by other peers.

Rendezvous Peers

Taken literally, a rendezvous is a gathering or meeting place; in P2P, a rendezvous peer provides peers with a network location to use to discover other peers and peer resources. Peers issue discovery queries to a rendezvous peer, and the rendezvous provides information on the peers it is aware of on the network. How a rendezvous peer discovers other peers on its local network will be discussed in the section "P2P Communication," later in this chapter.

A rendezvous peer can augment its capabilities by caching information on peers for future use or by forwarding discovery requests to other rendezvous peers. These schemes have the potential to improve responsiveness, reduce network traffic, and provide better service to simple peers.

A rendezvous peer will usually exist outside a private internal network's firewall. A rendezvous could exist behind the firewall, but it would need to be capable of traversing the firewall using either a protocol authorized by the firewall or a router peer outside the firewall.

Router Peers

A router peer provides a mechanism for peers to communicate with other peers separated from the network by firewall or Network Address Translation (NAT) equipment. A router peer provides a go-between that peers outside the firewall can use to communicate with a peer behind the firewall, and vice versa. This technique of firewall and NAT traversal is discussed in detail in the upcoming section "Challenges to Direct Communication."

To send a message to a peer via a router, the peer sending the message must first determine which router peer to use to communicate with the destination peer. This routing information provides a mechanism in P2P to replace traditional DNS, enabling an intermittently connected device with a dynamic IP address to be found on the network. In a similar manner to the way that DNS translates a simple name to an IP address, routing information provides a mapping between a unique identifier specifying a remote peer on the network and a representation that can be used to contact the remote peer via a router peer.

In simple systems, routing information might consist solely of resolving an IP address and a TCP port for a given unique identifier. A more complex system might provide routing information consisting of an ordered list of router peers to use to properly route a message to a peer. Routing a message through multiple router peers might be necessary to allow two peers to communicate by using a router peer to translate between two different and incompatible network transports.

Peer Groups

Before JXTA, the proprietary and specialized nature of P2P solutions and their associated protocols divided the usage of the network space according to the application. If you wanted to perform file sharing, you probably used the Gnutella protocol and could communicate only with other peers using the Gnutella protocol; similarly, if you wanted to perform instant messaging, you used ICQ and could communicate only with other peers also using ICQ.

The protocols' incompatibilities effectively divided the network space based on the application being used by the peers involved. If you consider a P2P system in which all

clients can speak the same set of protocols, as they can in JXTA, the concept of a peer group is necessary to subdivide the network space. As you would probably expect, a *peer group* is defined as follows:

A set of peers formed to serve a common interest or goal dictated by the peers involved. Peer groups can provide services to their member peers that aren't accessible by other peers in the P2P network.

Peer groups divide the P2P network into groups of peers with common goals based on the following:

- **The application they want to collaborate on as a group.** A peer group is formed to exchange service that the members do not want to have available to the entire population of the P2P network. One reason for doing this could be the private nature of the data used by the application.

- **The security requirements of the peers involved.** A peer group can employ authentication services to restrict who can join the group and access the services offered by the group.

- **The need for status information on members of the group.** Members of a peer group can monitor other members. Status information might be used to maintain a minimum level of service for the peer group's application.

Peer group members can provide redundant access to a service, ensuring that a service is always available to a peer group as long as at least one member is providing the service.

Network Transport

To exchange data, peers must employ some type of mechanism to handle the transmission of data over the network. This layer, called the *network transport*,is responsible for all aspects of data transmission, including breaking the data into manageable packets, adding appropriate headers to a packet to control its destination, and in some cases, ensuring that a packet arrives at its destination. A network transport could be a low-level transport, such as UDP or TCP, or a high-level transport, such as HTTP or SMTP.

The concept of a network transport in P2P can be broken into three constituent parts:

- **Endpoints—** The initial source or final destination of any piece of data being transmitted over the network. An endpoint corresponds to the network interfaces used to send and receive data.

- **Pipes—** Unidirectional, asynchronous, virtual communications channels connecting two or more endpoints.

- **Messages—** Containers for data being transmitted over a pipe from one endpoint to another.

To communicate using a pipe, a peer first needs to find the endpoints, one for the source of the message and one for each destination of the message, and connect them by binding a pipe to each of the endpoints. When bound this way, the endpoint acting as a data source is called an *output pipe* and the endpoint acting as a data sink is called an *input pipe*. The pipe itself isn't responsible for actually carrying data between the endpoints; it's merely an abstraction used to represent the fact that two endpoints are connected. The endpoints themselves provide the access to the underlying network interface used for transmitting and receiving data.

To send data from one peer to another, a peer packages the data to be transmitted into a message and sends the message using an output pipe; on the opposite end, a peer receives a message from an input pipe and extracts the transmitted data.

Notice that a pipe provides communication in only one direction, thus requiring two pipes to achieve two-way communication between two peers. The definition of a pipe is structured this way to capture the lowest common denominator possible in network communications, to avoid excluding any possible network transports. Although bidirectional communication is the norm in modern networks, there's no reason to exclude the possibility of a unidirectional communications channel in the definition because any bidirectional network transport can easily be modeled using two unidirectional pipes.

Services

Services provide functionality that peers can engage to perform "useful work" on a remote peer. This work might include transferring a file, providing status information, performing a calculation, or basically doing anything that you might want a peer in a P2P network to be

capable of doing. Services are the motivation for gathering devices into a P2P network; without services, you don't a have a P2P network—you have just a set of devices incapable of leveraging each other's resources.

Services can be divided into two categories:

- **Peer services—** Functionality offered by a particular peer on the network to other peers. The capabilities of this service will be unique to the peer and will be available only when the peer is connected to the network. When the peer disconnects from the network, the service is no longer available.

- **Peer group services—** Functionality offered by a peer group to members of the peer group. This functionality could be provided by several members of the peer group, thereby providing redundant access to the service. As long as one member of the peer group is connected to the network and is providing the service, the service is available to the peer group.

Most of the functionality required to create and maintain a P2P network, such as the underlying protocols required to find peers and resources, could also be considered services. These *core services* provide the basic P2P foundation used to build other, more complex services.

Advertisements

Until now, P2P applications have used an informal form of advertisements. In Gnutella, the results returned by a search query could be considered an advertisement that specifies the location of a specific song file on the Gnutella network. These primitive advertisements are extremely limited in their purpose and application. At its core, an *advertisement* is defined as follows:

A structured representation of an entity, service, or resource made available by a peer or peer group as a part of a P2P network.

All the building blocks discussed up to this point in the chapter can be described by advertisements, including peers, peer groups, pipes, endpoints, services, and content. When you start looking at advertisements in JXTA, you'll see the power of describing

resources as advertisements and learn how advertisements simplify the task of organizing P2P networks.

Protocols

Every data exchange relies on a protocol to dictate what data gets sent and in what order it gets sent. Even the simplest human gesture, the handshake, is built on a protocol that defines when it's appropriate to shake hands, which hand to use, and how long to shake. A *protocol* is simply this:

A way of structuring the exchange of information between two or more parties using rules that have previously been agreed upon by all parties.

In P2P, protocols are needed to define every type of interaction that a peer can perform as part of the P2P network:

- Finding peers on the network
- Finding what services a peer provides
- Obtaining status information from a peer
- Invoking a service on a peer
- Creating, joining, and leaving peer groups
- Creating data connections to peers
- Routing messages for other peers

The organization of information into advertisements simplifies the protocols required to make P2P work. The advertisements themselves dictate the structure and representation of the data, simplifying the definition of a protocol. Rather than passing back and forth raw data, protocols simply organize the exchange of advertisements containing the required information to perform some arbitrary functionality.

Entity Naming

Most items on a P2P network need some piece of information that uniquely identifies them on the network:

- **Peers—** A peer needs an identifier that other peers can use to locate or specify it on the network. Identifying a particular peer could be necessary to allow a message to be routed through a third party to the correct peer.

- **Peer groups—** A peer needs some way to identify which peer group it would like to use to perform some action. Actions could include joining, querying, or leaving a peer group.

- **Pipes—** To permit communication, a peer needs some way of identifying a pipe that connects endpoints on the network.

- **Contents—** A piece of content needs to be uniquely identifiable to enable peers to mirror content across the network, thereby providing redundant access. Peers can then use this unique identifier to find the content on any peer.

In traditional P2P networks, some of these identifiers might have used network transport-specific details; for example, a peer could be identified by its IP address. However, using system-dependent representations is inflexible and can't provide a system of identification that is independent of the operating system or network transport. In the ideal P2P network, any device should be capable of participating, regardless of its operating system or network transport. A system-independent entity naming scheme is a requirement for a flexible P2P network.

P2P Communication

The fundamental problem in P2P is how to enable the exchange of services between networked devices. Solving this problem requires first finding answers to two important questions:

- How does a device find peers and services on a P2P network?

- How does a device in a private network participate in P2P?

The first question is important because, without the knowledge of the existence of a peer or a service on the network, there's no possibility for a device to engage that service. The second question is important to answer because many devices in a P2P network will be separated from the network at large by networking equipment designed to prevent or restrict direct connections between two devices in different internal private networks.

Finding Advertisements

Any of the basic building blocks discussed in the last section can be represented as an advertisement, and that characteristic considerably simplifies the problem of finding peers, peer groups, services, pipes, and endpoints. Instead of worrying about the specific case, such as finding a peer, you need to consider only the general problem of finding advertisements on the network.

A peer can discover an advertisement in three ways:

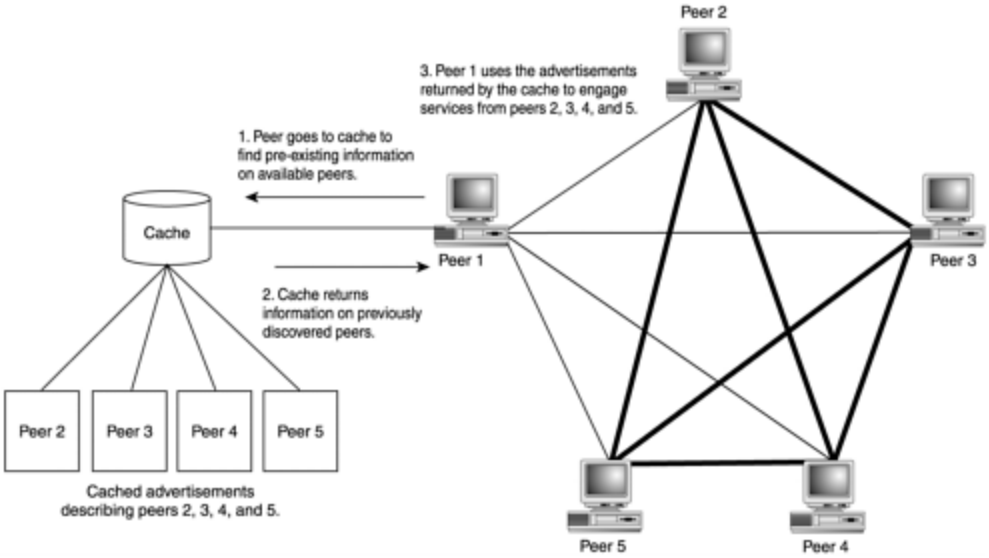
- No discovery
- Direct discovery
- Indirect discovery

The first technique involves no network connectivity and can be considered a passive discovery technique. The other two techniques involve connecting to the network to perform discovery and are considered active discovery techniques.

No Discovery

The easiest way for a peer to discover advertisements is to eliminate the process of discovery entirely. Instead of actively searching for advertisements on the network, a peer can rely on a cache of previously discovered advertisements to provide information on peer resources, as shown in Figure 2.1. Although this method might sound trivial, it can effectively reduce the amount of network traffic generated by the peer and allow a peer to obtain nearly instantaneous results, unlike active discovery methods.

Figure 2.1. Peer discovery using cached advertisements.



In its simplest form, the local cache might consist only of a text file that lists the IP addresses and ports of previously discovered rendezvous peers, thereby providing a starting point for active peer discovery. At the other extreme, a cache might be as comprehensive as a database of every advertisement discovered by the peer in the past. The cache of advertisements might even be hard-coded into the P2P application itself, although this would limit the flexibility of the application somewhat.

A drawback of using a cache of known advertisements is the potential for advertisements in the cache to grow *stale* and describe resources that are no longer available on the network. This presents a problem when a peer attempts to engage a resource described by a stale advertisement and fails to engage the service. Although the cache has the potential to reduce network traffic, in this case, stale advertisements in the cache increase network traffic. When a peer attempts to engage a resource over the network and discovers that the resource is no longer available, the peer will probably have to resort to an active discovery method. Thus, the peer engages the network twice in this case instead of once, which would have been the case if it had used only active discovery.

To reduce the possibility that a given advertisement is stale, a cache can expire advertisements, thereby removing them from the cache based on the probability that a given advertisement is still valid.

One way to expire advertisements is to store a *best before* timestamp in the cache with each advertisement. When an advertisement is discovered, a timestamp is stored in the cache, setting the maximum lifespan of the advertisement. Before using an advertisement, the cache checks the advertisement's best before timestamp and discards the advertisement if it's no longer considered valid. Instead of waiting for an advertisement to be used, the cache might also periodically cull the store of expired advertisements to reduce storage requirements and improve responsiveness.

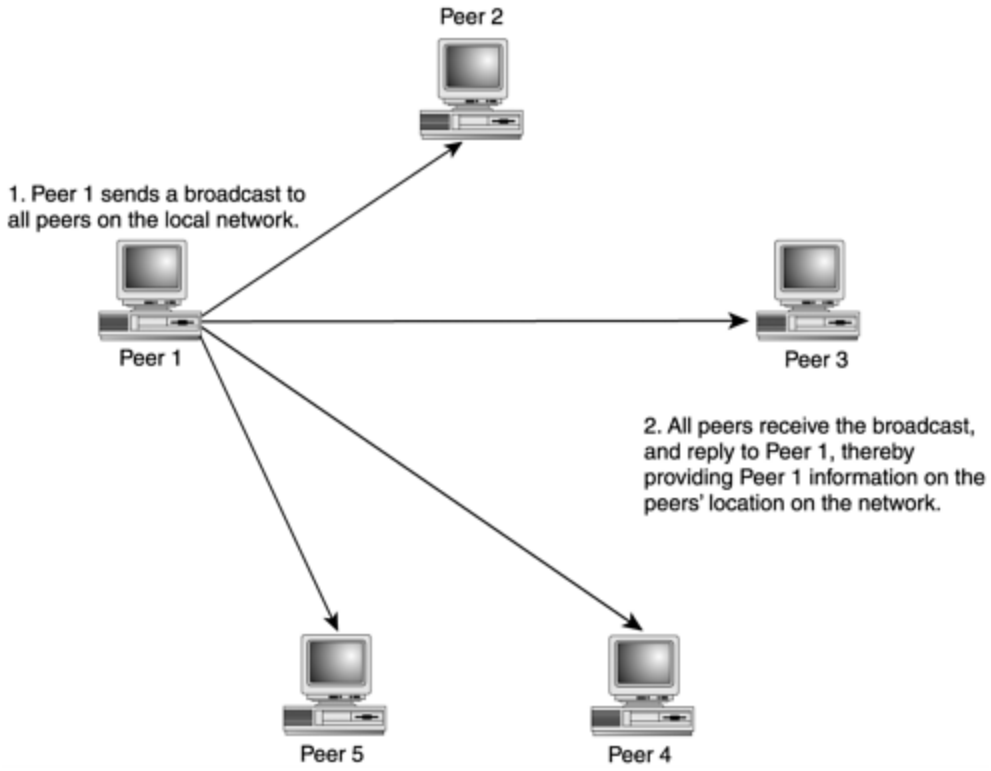
Another expiration technique that a cache might use is a first-in, first-out stack of advertisements with a fixed maximum size for the stack. When the cache is full, adding a new advertisement to the stack pushes out the oldest advertisement first.

Using a cache to discover advertisements is simple to implement, especially when built in conjunction with active discovery methods. In most modern programming languages, it's trivial to create code that processes an advertisement from an abstract source and to create wrappers for file and network sources. When done this way, the code is independent of the source and will operate the same regardless of whether the advertisement originated from a file cache or from another peer on the network.

Direct Discovery

Peers that exist on the same LAN might be capable of discovering each other directly without relying on an intermediate rendezvous peer to aid the discovery process. Direct discovery requires peers to use the broadcast or multicasting capabilities of their native network transport, as shown in Figure 2.2.

Figure 2.2. Direct peer discovery.



When other peers have been discovered using this mechanism, the peer can discover other advertisements by communicating directly with the peers, without using broadcast or multicast capabilities.

Unfortunately, this discovery technique is limited to peers located on the same local LAN segment and usually can't be used to discover peers outside the local network. Discovering peers and advertisements outside the private network requires indirect discovery conducted via a rendezvous peer.

Indirect Discovery

Indirect discovery requires using a rendezvous peer to act as a source of known peers and advertisements, and to perform discovery on a peer's behalf. This technique can be used by peers on a local LAN to find other peers without using broadcast or multicast capabilities, or by peers in a private internal network to find peers outside the internal network.

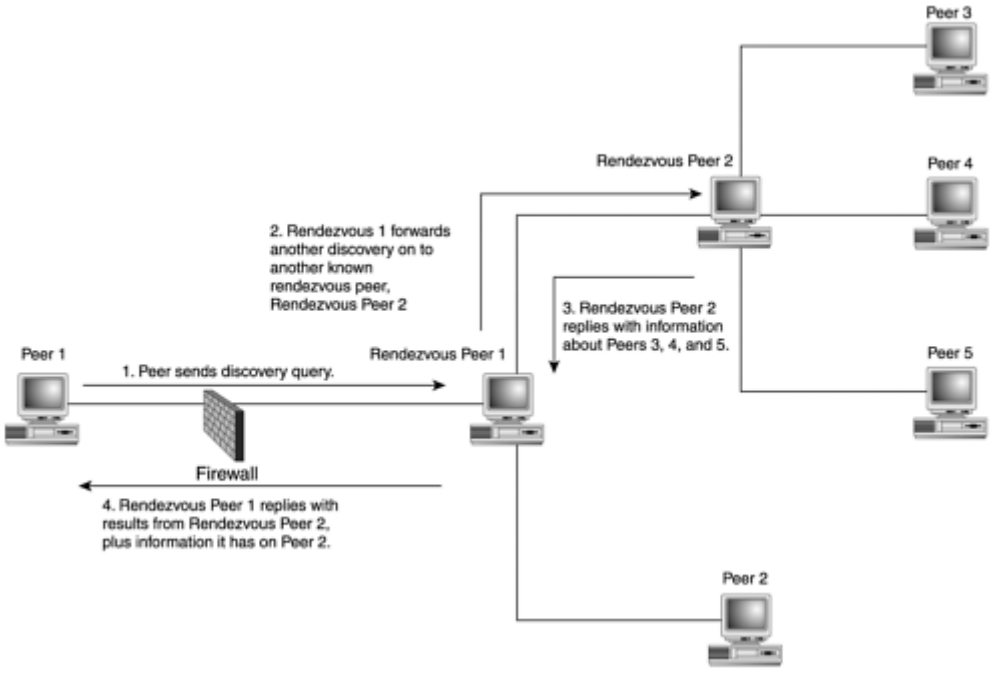
Rendezvous peers provide peers with two possible ways of locating peers and other advertisements:

- **Propagation—** A rendezvous peer passes the discovery request to other peers on the network that it knows about, including other rendezvous peers that also propagate the request to other peers.

- **Cached advertisements—** In the same manner that simple peers can use cached advertisements to reduce network traffic, a rendezvous can use cached advertisements to respond to a peer's discovery queries.

When used together as shown in Figure 2.3, propagation and caching provide an effective solution for rendezvous peers to cache a large number of advertisements and serve a large number of simple peers. As each simple or rendezvous peer responds to the discovery request, the rendezvous peer can cache the response for future use, further reducing network traffic and increasing network performance.

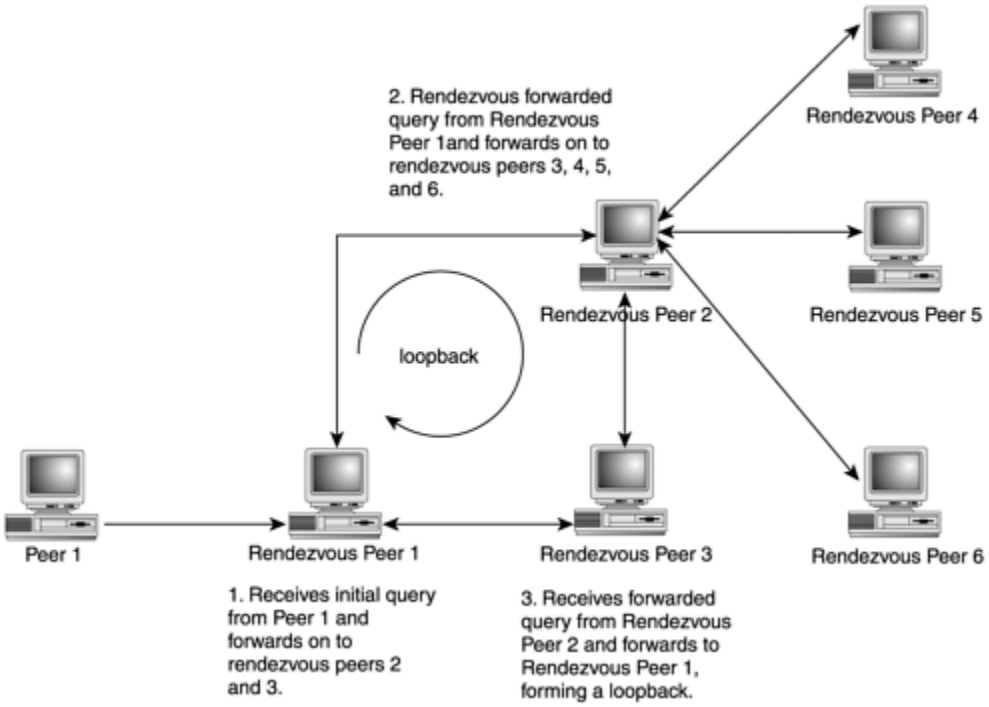
Figure 2.3. Indirect discovery via a rendezvous peer.



Although caching reduces network traffic required to discover resources, propagating discovery queries to other rendezvous peers without restriction can lead to severe network congestion on a P2P network, as shown in Figure 2.4. When one rendezvous receives a

discovery query, it forwards the request to all the rendezvous peers that it knows; one query comes in, and many queries go out.

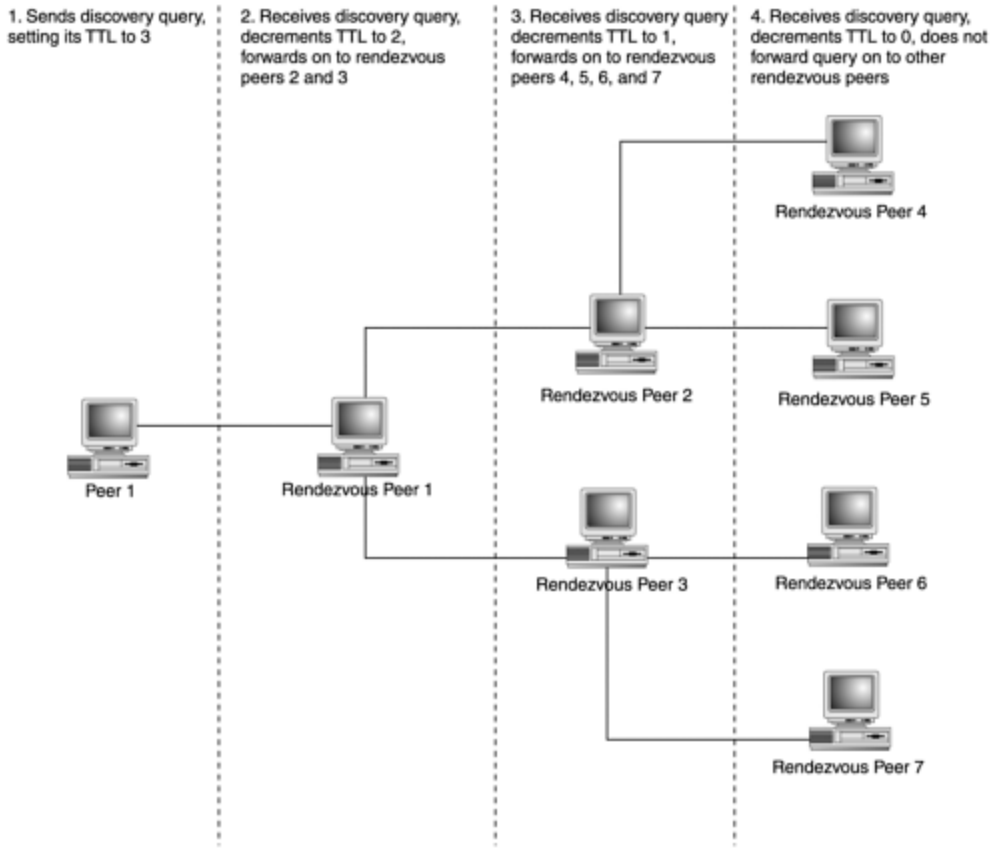
Figure 2.4. Discovery propagation chaos.



This retransmission *amplifies* the discovery query. When the query is propagated to other rendezvous peers, it is amplified again, dramatically increasing the load on the network. Adding to the problem of unchecked propagation, a discovery query's path could double back on itself, creating a feedback loop or *loopback* in the network.

To prevent excessive propagation of requests, messages usually incorporate a *Time To Live* (TTL) attribute. TTL is expressed as the maximum number of times a query should be propagated between peers on the network. As shown in Figure 2.5, when a rendezvous peer receives a message containing a discovery query, it decrements the message's TTL by 1 and discards the query if the resulting TTL value is 0. Otherwise, the query message is propagated to other peers using the new TTL value.

Figure 2.5. Illustration of TTL in discovery propagation.



As a result, each message has a maximum *radius* on the network that it can travel. Of course, for this technique to work, all rendezvous peers must properly decrement the TTL field.

To address the problem of loopback, propagated messages can include path information along with the request. Rendezvous peers along the way can use this path information to prevent propagating a message to a rendezvous that has already received the message. Although this technique eliminates loopback, it doesn't prevent a rendezvous peer from getting the same message multiple times through indirect paths.

Discovering Rendezvous and Routing Peers

For most peers existing on a private internal network, finding rendezvous and router peers is critical to participating in the P2P network. Because of the restrictions of a private network's firewall, a peer on an internal network has no capability to use direct discovery

to perform discovery outside the internal network. However, a peer might still be capable of performing indirect discovery using rendezvous and router peers on the internal network.

In most P2P applications, the easiest way to ensure that a simple peer can find rendezvous and router peers is to seed the peer with a hard-coded set of rendezvous and router peers. These rendezvous and router peers usually exist at static, resolvable IP addresses and are used by a peer as an entrance point to the P2P network. A peer located behind a firewall can use these static rendezvous peers as a starting point for discovering other peers and services and can connect to other peers using the static set of router peers to traverse firewalls.

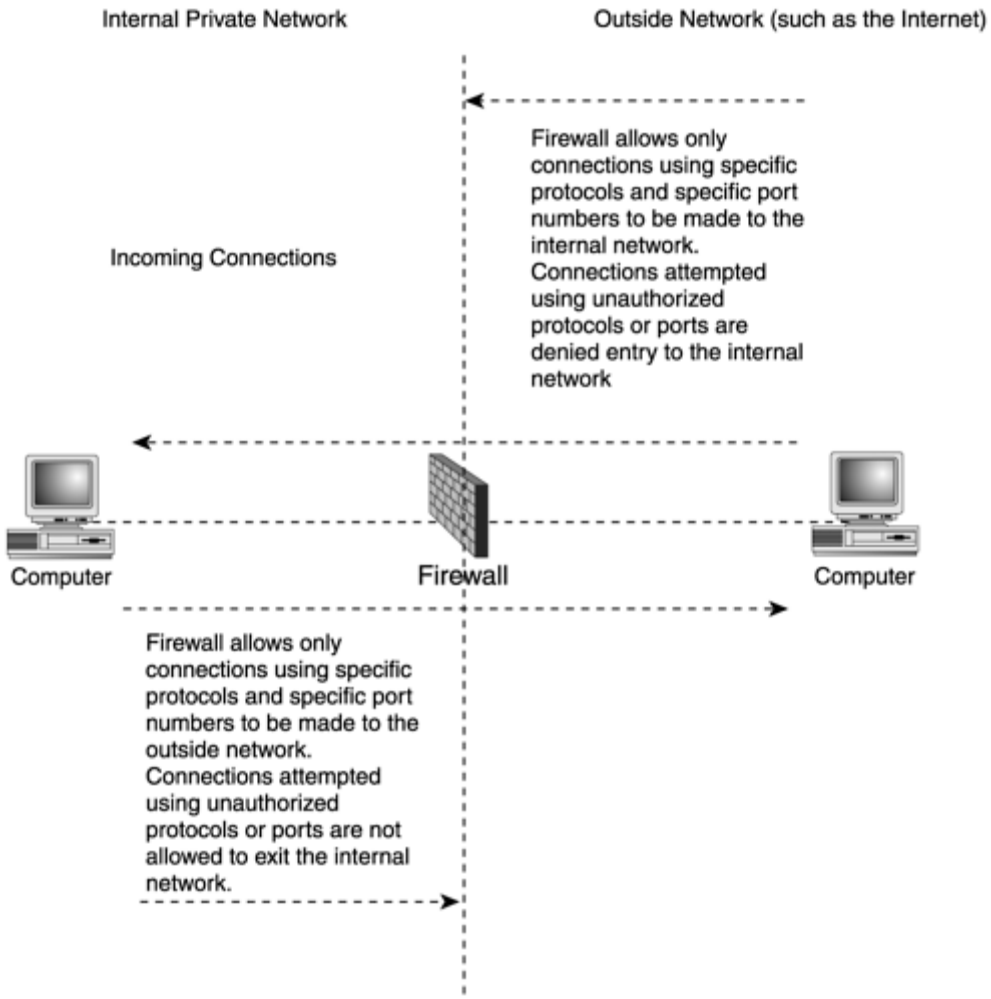
Challenges to Direct Communication

The use of firewalls and NAT by corporate private networks poses a serious obstacle to P2P networking. NAT and firewalls are usually used together to secure a corporate network against unauthorized network activity originating from either inside or outside the network and to provide a private internal networking environment.

Firewalls

Firewalls are used to protect corporate networks from unauthorized network connections, either incoming from the outside network or outgoing from the internal network, as shown in Figure 2.6. Typically firewalls use IP filtering to regulate which protocols may be used to connect from outside the firewall to the internal network or vice versa. A firewall might also regulate the ports used by outside clients to initiate inbound connections to the internal network or by internal clients to initiate outbound connections from the internal network.

Figure 2.6. A network topology using a firewall.



Because a firewall might block incoming connections, a peer outside the firewall will most likely not be capable of connecting directly to a peer inside the firewall. A peer within the network might also be restricted to using only certain protocols (such as HTTP) to connect to locations outside the firewall, further limiting the types of P2P communication possible.

Network Address Translation (NAT)

NAT is a technique used to map a set of private IP addresses within an internal network to another set of external IP addresses on a public network. NAT comes in two varieties:

- **Static NAT—** In static NAT, the mapping relationship between internal and external IP addresses is one-to-one. Every internal IP address is mapped to one and only one external IP address.

- **Dynamic NAT—** Dynamic NAT maps the set of internal IP addresses to a smaller set of external IP addresses.

A private network employing NAT usually assigns internal IP addresses from one of the ranges of IP addresses defined specifically for private networks:

- Class A private addresses: 10.0.0.0 through 10.255.255.255

- Class C private addresses: 192.168.0.0 through 192.168.255.255

- Class B private addresses: 172.16.0.0 through 172.31.255.255

A machine using an IP address within this range is most likely behind NAT equipment.

NAT is used for a variety of reasons, the most popular reason being that it eliminates the need for global unique IP addresses for every workstation within a corporation, thereby reducing the cost of a corporate network. NAT also enables system administrators to protect a network by providing only a single point of entry into the internal network. NAT accomplishes this by allowing only incoming connections to internal machines that originally initiated a connection to the outside network. Rather than attempting to protect each machine using a firewall to filter incoming connections, a system administrator can use NAT to ensure that the only connections allowed back into the network are those that originated within the network.

NAT is usually implemented by a router or a firewall acting as a gateway to the Internet for the private internal network. To map a packet from an internal IP address to an external IP address, the router does the following:

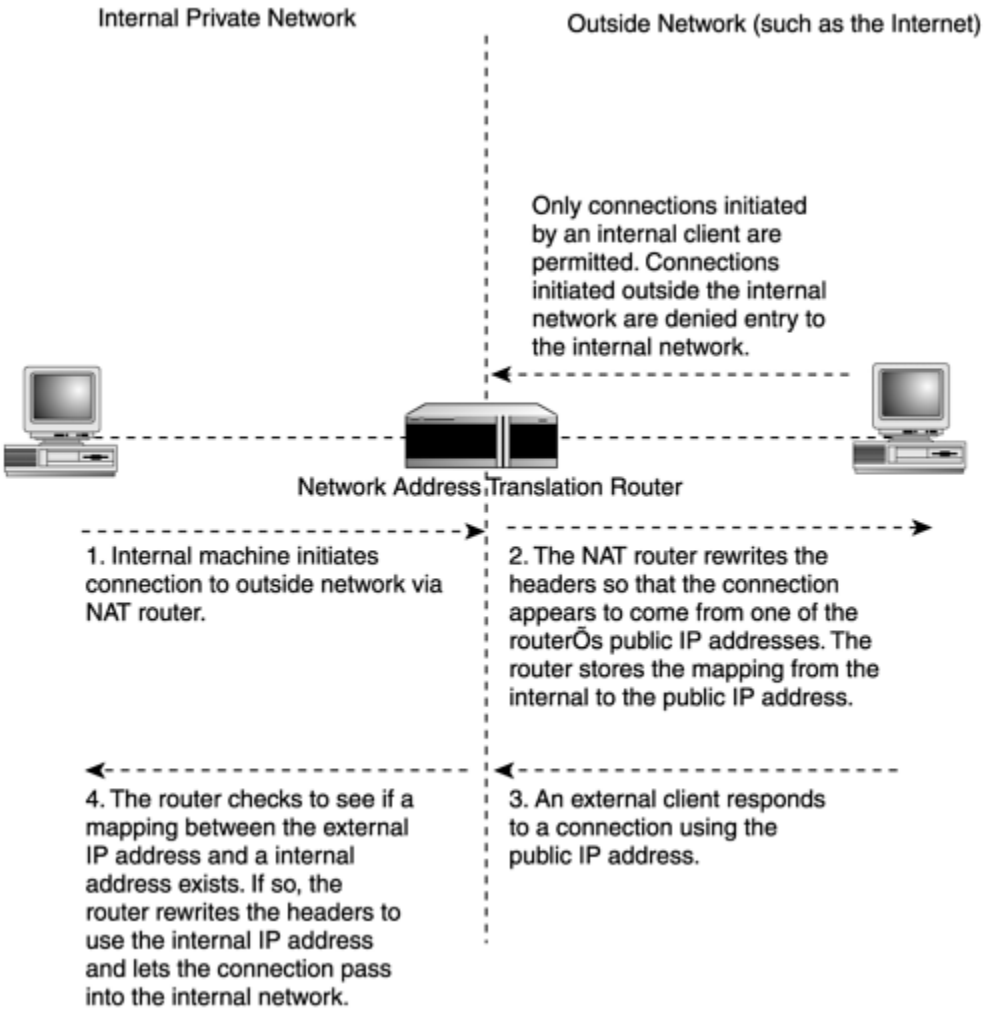
1. Stores the source IP address and port number of the packet in the router's translation table

2. Replaces the source IP address for the packet with one of the IP addresses from the router's pool of public IP addresses, storing the mapping of the original IP address to the public IP address in the translation table in the process

3. Replaces the source port number with a new port number that it assigns and stores the mapping in the translation table

After each step has been performed, the packet is forwarded to the external network. Data packets arriving at one of the router's external public IP addresses go through an inverse mapping process that uses the router's translation table to map the external port number and IP address to an internal IP address and port number. If no matching entry for a given public IP address and port number is found in the translation table, the router blocks the data from entering the internal private network. The flow of data across a NAT router is illustrated in Figure 2.7.

Figure 2.7. A network topology using Network Address Translation.



NAT protects networks by allowing only connections to the internal network that originated within the internal network. A machine outside the network can't connect to a machine in the internal network unless the internal machine initiated the connection to the external machine. As a result, an external peer in a P2P network has no mechanism to spontaneously connect to a peer located behind a NAT gateway. From the outside peer's point of view, the peer doesn't exist because no mapping between external and internal IP addresses and port numbers exists in the router's translation table.

Traversing the NAT/Firewall Boundary

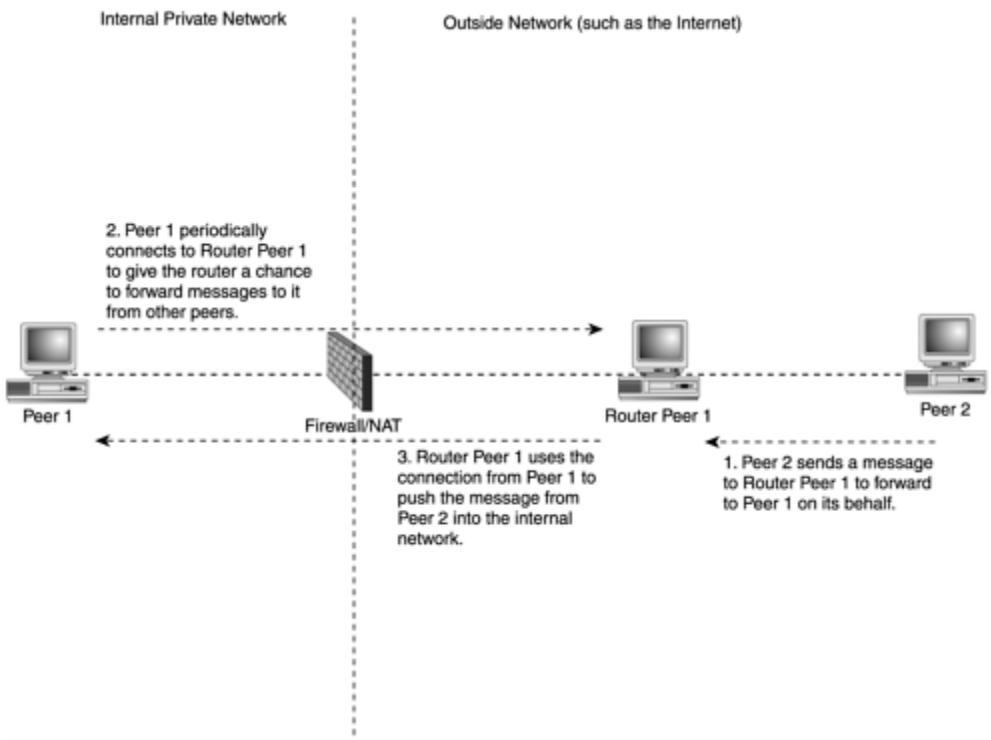
The combined use of NAT and firewalls results in an especially difficult set of circumstances for peer communication: Peers can't connect to machines behind NAT unless the internal peer initiates communication, and connections can be blocked at the firewall based on the connection's protocol or destination IP address and port number.

The only tool that a peer has at its disposal to solve this problem is its capability to create outgoing network connections to hosts outside the firewall/NAT gateway. Peers can use protocols permitted by the firewall to tunnel connections through the firewall to the outside network. By initiating the connection within the internal network, the necessary mapping in the NAT router translation tables is set up, allowing an external machine to send data back into the internal network. However, if a firewall is configured to deny all outgoing connections, peer communication is impossible.

In most corporate networks, HTTP is the protocol most likely to be enabled by a firewall for outgoing connections. Unfortunately, HTTP is a request-response protocol: Each HTTP connection sends a request and then expects a response. The connection must remain open after the initial request to receive the response. Although HTTP provides a peer with a mechanism to send requests out of the internal network, it doesn't provide the capability for external peers to spontaneously cross the firewall boundary to connect to peers inside the internal network.

To address this problem, a peer inside a firewall uses a router peer either located outside the firewall or visible outside the firewall to traverse the firewall, as shown in Figure 2.8. Peers attempting to contact a peer behind a firewall connect to the router peer, and the peer behind the firewall periodically connects to a router peer. When the internal peer connects to the router, any incoming messages get *pushed* down to the peer in the HTTP response.

Figure 2.8. Traversing a firewall/NAT.



This technique can be used with any protocol permitted by the firewall and understood by the router peer. The router peer effectively translates between the network transport used for P2P communication and the transport used to tunnel through the firewall.

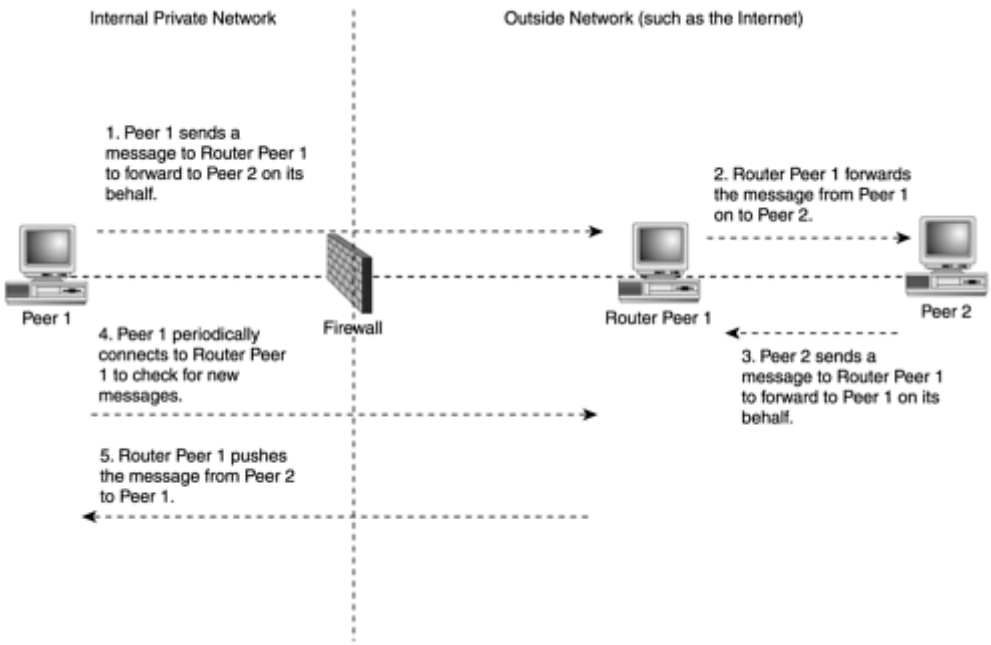
Routing Messages Between Peers

In cases when a firewall or NAT is located between two peers, a router peer must be used to proxy a connection between the public network and the peer located inside the firewall. In the simple case, only a single firewall separates the source and destination peers, thus requiring only a single router peer. In more complex cases, a firewall or NAT can protect each of the peers and require the use of multiple router peers to traverse each firewall/NAT boundary.

Single Firewall/NAT Traversal

Figure 2.9 shows the process for sending messages outside a single firewall/NAT.

Figure 2.9. Outgoing single firewall/NAT traversal.



To allow a peer located inside a firewall/NAT to send a message to another peer located on the public network, three steps are required:

1. The peer behind the firewall/NAT connects to the router peer using a protocol capable of traversing the firewall, such as HTTP, and requests that the router peer forward a message to a destination peer.

2. The router accepts the connection from the peer behind the firewall and initiates a connection to the requested destination on the peer's behalf. This connection uses whatever network transport both the router peer and the destination peer have in common.

3. The message is sent from the source to the destination peer by the router peer, acting as a proxy for the source peer.

After the message from the source peer has been sent to the destination peer, the connection closes. Further messages can be sent by repeating the procedure, but the message might use a different router peer and, therefore, might follow a different route to the destination peer.

To allow a public peer to send a message to a peer located behind a firewall/NAT, the source peer must know routing information that describes a router peer capable of routing the message to the destination peer. Route information might have been obtained previously during discovery or might require an additional discovery request to the P2P network. When the source peer has obtained routing information, sending the message involves three steps:

1. The source peer opens a connection to the router peer, asking it to forward the message on to the destination peer.

2. The router peer waits until the destination peer connects to it using a protocol capable of traversing the firewall, such as HTTP.

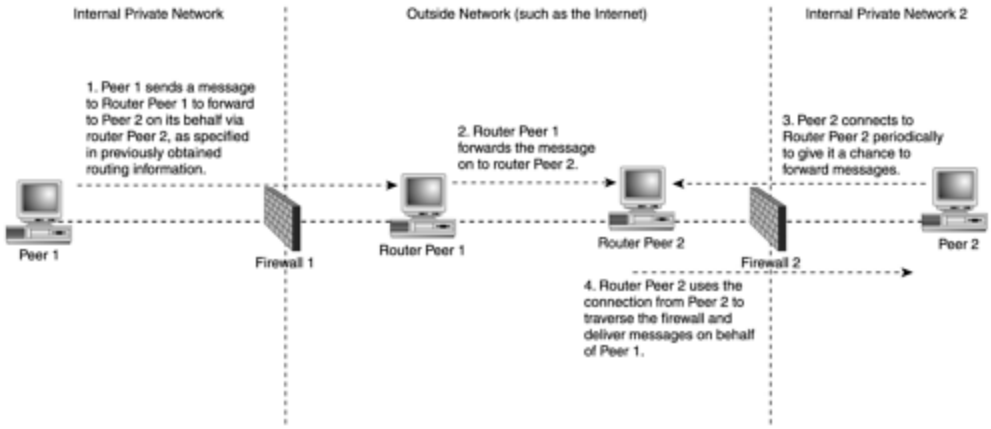
3. The destination peer connects to the router peer periodically, at which point the message is pushed down to the destination peer.

Again, when the message reaches the destination peer, the connection between the router peer and the other two peers is closed. Sending another message from the source peer requires repeating the procedure and might use a different router peer to provide connectivity to the destination peer.

Double Firewall/NAT Traversal

Most simple peers located at the *edge* of the Internet are likely to be protected by a firewall/NAT, so any message being sent from a source peer to a destination peer will need to traverse two firewall/NAT boundaries. The procedure for traversing two firewalls is similar to the single firewall traversal case and basically combines both the incoming and the outgoing cases of the single firewall traversal scenario. Figure 2.10 illustrates a double firewall/NAT traversal.

Figure 2.10. Double firewall traversal.



Before a source peer can send the message, it needs to locate routing information for the peer that describes a set of router peers capable of proxying messages to the destination peer. In this case, more than one router peer might be involved; one router peer is required to allow the source peer to traverse its firewall, and another is required to traverse the firewall providing access to the destination peer. When the source peer has this routing information, sending the message involves four steps:

1. The source peer opens a connection to the source router peer, asking it to forward the message on to the destination peer by way of the destination router peer provided.

2. The source router peer opens a connection to the destination router peer. This connection uses whatever network transport both router peers have in common.

3. The destination router peer waits until the destination peer connects to it using a protocol capable of traversing the firewall, such as HTTP.

4. The destination peer connects to the router peer periodically, and the message is pushed down to the destination peer.

Traversing both firewalls might involve only one router peer if both the source and the destination peers have a router peer in common; however, traversing firewall boundaries isn't the only reason to use a router peer. Multiple router peers can be used by a peer to circumnavigate network bottlenecks and achieve greater performance, or to provide translation between two incompatible networks transports. When the peer connects to the

source router peer in this case, it provides an ordered list of router peers to use to send the message to the peer on its behalf.

Comparisons to Existing P2P Solutions

Using the building blocks of P2P networks defined in this chapter, it's possible to interpret existing proprietary P2P solutions, such as Napster and Gnutella, or even non-P2P applications, such as the client/server architecture.

Napster

Napster's hybrid P2P network, consisting of a centralized server for performing search functionality, could be modeled as a single rendezvous peer and multiple simple peers, all using TCP as a network transport. The rendezvous peer provides simple peers with the capability to locate an MP3 file advertisement consisting of filename, IP address, and port information. Simple peers use this information to connect directly and download the file from its host peer.

Napster doesn't provide a complete solution for bypassing firewalls, and it is capable of traversing only a single firewall. Each peer acts as a simple router, capable of sending content to a firewalled peer when a request is made via HTTP. Napster provides no message-routing capabilities, meaning that simple peers on the network can't act as router peers to enable other peers to perform double firewall traversal.

Gnutella

In the Gnutella network, each peer acts as a simple peer, a rendezvous peer, and a router peer, 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, which then propagate the query to other peers. Advertisements for content on the Gnutella network consist of an IP address, a port number, an index number identifying the file on the host peer, and file details such as name and size. Gnutella peers don't provide full router peer capabilities, which means that, as with Napster, Gnutella peers are capable of traversing only a single firewall.

Client/Server

Even traditional client/server architecture can be interpreted in terms of the P2P building blocks. The client acts as a simple peer, and the server acts as a rendezvous peer capable of providing advertisements that vary according to the application. No capabilities for traversing firewalls or NAT are provided, and the network transport used varies by application.

The definitions of these basic P2P building blocks will be expanded in the coming chapters to incorporate the implementation-specific details defined by JXTA and the Java reference implementation of JXTA.

Summary

This chapter presented the basic building blocks of P2P networking and explained some of the obstacles that a P2P network must overcome. Specifically, this chapter explained the barrier to P2P communication presented by firewall/NAT routers, provided background information on how they work, and explained how P2P manages to provide connectivity to private networks protected by firewall/NAT routers.

The next chapter builds on this chapter and reveals the JXTA realization of the building blocks defined in this chapter. Using the JXTA Shell, you'll see how to experiment with these primitives directly, to better understand them before you explore the JXTA protocols.

Chapter 3. Introducing JXTA P2P Solutions

Now that you've gotten a basic introduction to the terminology, components, and issues of P2P networking, it's time to begin exploring the JXTA platform. This chapter introduces the logical structure and building blocks of JXTA, and demonstrates the capabilities of JXTA using the JXTA Shell application to provide interactive experimentation with the JXTA platform.

As outlined in Chapter 2, "P2P Concepts," a complete P2P solution provides mechanisms for a peer to do the following:

- Discover other peers and their services
- Publish its available services
- Exchange data with another peer
- Route messages to other peers
- Query peers for status information
- Group peers into peer groups

The JXTA platform defines a set of protocols designed to address the common functionality required to allow peers on a network to form robust pervasive networks, independent of the operating system, development language, and network transport employed by each peer.

Core JXTA Design Principles

While designing the protocol suite, the Project JXTA team made a conscious decision to design JXTA in a manner that would address the needs of the widest possible set of P2P applications. The design team stripped the protocols of any application-specific assumptions, focusing on the core P2P functionality that forms the foundation of all types of P2P applications.

One of the most important design choices was not to make assumptions about the type of operating system or development language employed by a peer. By making this choice, the Project JXTA team hoped to enable the largest number of potential participants in any JXTA-enabled P2P networking application. The JXTA Protocols Specification expressly states that network peers should be assumed to be any type of device, from the smallest embedded device to the largest supercomputer cluster.

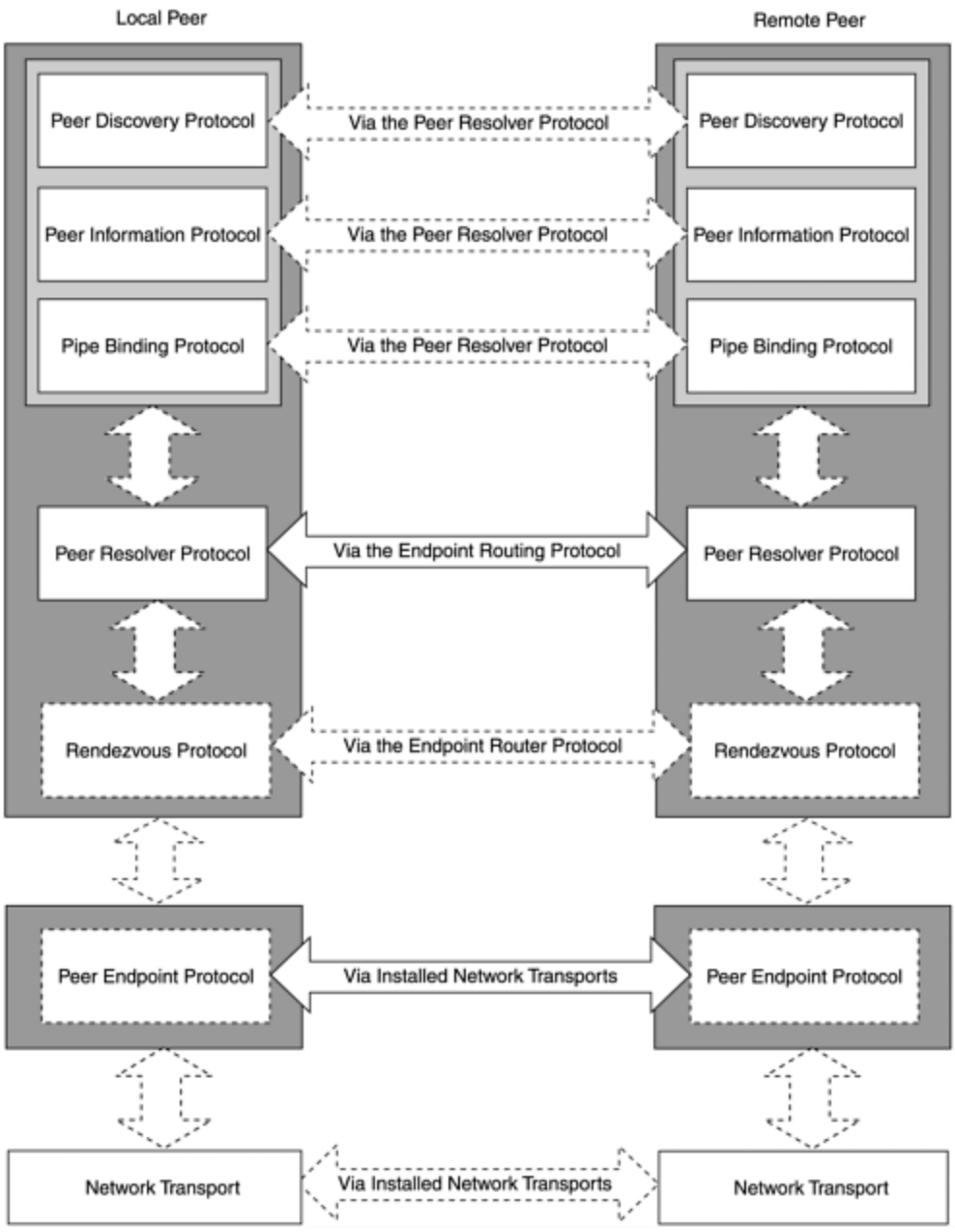
In addition to eliminating barriers to participation based on operating system, computing platform, or programming language, JXTA makes no assumptions about the network transport mechanism, except for a requirement that JXTA must not require broadcast or multicast transport capabilities. JXTA assumes that peers and their resources might appear and disappear spontaneously from the network and that a peer's network location might change spontaneously or be masked by Network Address Translation (NAT) or firewall equipment.

Apart from the requirements specified by the JXTA Protocols Specification, the specification makes several important recommendations. In particular, the specification recommends that peers cache information to reduce network traffic and provide message routing to peers that are not directly connected to the network.

The JXTA Protocol Suite

Based on these design criteria and others documented in the Protocols Specification, the Project JXTA team designed a set of six protocols based on XML messages, shown in Figure 3.1.

Figure 3.1. The JXTA protocol stack.



Each of the JXTA protocols addresses exactly one fundamental aspect of P2P networking. Each protocol conversation is divided into a portion conducted by the local peer and another portion conducted by the remote peer. The local peer's half of the protocol is responsible for generating messages and sending them to the remote peer. The remote peer's half of the protocol is responsible for handling the incoming message and processing the message to perform a task.

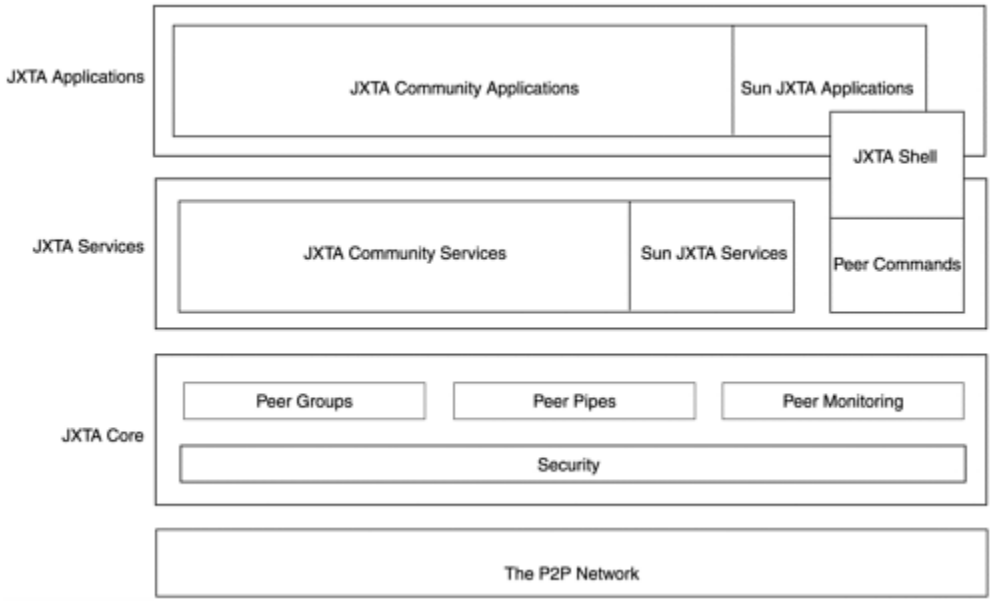
Each protocol is semi-independent of the others. A peer can elect to implement only a subset of the protocols to provide functionality, while relying on prespecified behavior to eliminate the need for a protocol. For example, a peer could rely on a preconfigured set of router peers and, therefore, would not require an implementation of the Endpoint Routing Protocol. However, the protocols aren't entirely independent of each other because each layer in the JXTA protocol stack depends on the layer below to provide connectivity to other peers. Although it would be possible to build an independent implementation of the Peer Discovery Protocol, it wouldn't be useful without an implementation of the Peer Resolver and Endpoint Routing Protocols to handle transporting its messages to remote peers.

Peers can even elect to implement only one half of a protocol to provide a peer optimized for one specific task. However, despite the allowance for partial implementations, the JXTA specification recommends that peers completely implement all the protocols.

The Logical Layers of JXTA

The JXTA platform can be broken into three layers, as shown in Figure 3.2.

Figure 3.2. The JXTA three-layer architecture.



Each layer builds on the capabilities of the layer below, adding functionality and behavioral complexity.

The Core Layer

The core layer provides the elements that are absolutely essential to every P2P solution. Ideally, the elements of this layer are shared by all P2P solutions. These concepts were discussed in Chapter 2. The elements of the core layer are listed here:

- Peers
- Peer groups
- Network transport (pipes, endpoints, messages)
- Advertisements
- Entity naming (identifiers)
- Protocols (discovery, communication, monitoring)
- Security and authentication primitives

The core layer includes the six main protocols provided by JXTA. Although these protocols are implemented as services, they are located in the platform layer and are designated as core services to distinguish them from the service solutions of the services layer.

The core layer, as its name suggests, is the fundamental core of the JXTA solution. All other aspects of a JXTA P2P solution in the services or applications layers build on this layer to provide functionality.

The Services Layer

The services layer provides network services that are desirable but not necessarily a part of every P2P solution. These services implement functionality that might be incorporated into several different P2P applications, such as the following:

- Searching for resources on a peer
- Sharing documents from a peer
- Performing peer authentication

The services layer encompasses additional functionality that is being built by the JXTA community (open-source developers working with Project JXTA) in addition to services built by the Project JXTA team. Services built on top of the JXTA platform provide specific capabilities that are required by a variety of P2P applications and can be combined to form a complete P2P solution.

The Applications Layer

The applications layer builds on the capabilities of the services layer to provide the common P2P applications that we know, such as instant messaging. Because an application might encompass only a single service or aggregate several services, it's difficult sometimes to determine what constitutes an application and what constitutes a service.

Usually, the presence of some form of user interface indicates an application rather than a service. In the case of the JXTA Shell, most of the functionality is built on peer commands, simple services that accept command-line arguments from the JXTA Shell. The JXTA Shell itself is a service, providing only a minimal user interface, so the Shell is spread across the application/service boundary.

Applications include those P2P applications being built by the JXTA Community, as well as demonstration applications such as the JXTA Shell being built by the Project JXTA team.

XML: A Brief Introduction

All aspects of JXTA build on the eXtensible Markup Language (XML) to structure data as advertisements, messages, and protocols. XML is good choice for representing data for five reasons:

- **XML is language-neutral.** Any programming language capable of manipulating text strings is capable of parsing and formatting XML data.

- **XML is simple.** XML uses text markup to structure data in much the same way that HTML structures text documents for display in web browsers. The simplicity of XML makes it easier for developers to understand and debug.

- **XML is self-describing.** An XML document consists of data structured using metadata tags and attributes that describe the format of the data. Although XML supports the use of Document Type Definitions (DTDs) to provide a schema definition of a valid document, this is not a requirement for a well-formed XML document.

- **XML is extensible.** Unlike HTML, XML allows authors to define their own set of markup tags to structure data.

- **XML is a standard.** The World Wide Web Consortium (www.w3.org) is responsible for maintaining the XML standard, with industry and community input, and has been widely adopted in all areas of the computer industry.

To learn all you'll need to know about XML to understand JXTA, consider the simple example given in Listing 3.1.

Listing 3.1 A Simple XML Example

```
<?xml version="1.0" encoding="UTF-8"?>
<Person>
  <Name>Erwin van der Koogh</Name>
  <Address>12 Lower Hatch Street</Address>
  <City>Dublin</City>
  <Country>Ireland</Country>
  <Phone>555-5555</Phone>
</Person>
```

Even if you've never seen XML, you probably recognize the example XML document as the contact information for a person named Erwin van der Koogh. From the example, you might guess at some of the rules of XML as follows:

- Each piece of information is encapsulated between a beginning and an end tag (such as `<Name></Name>`).
- The name of a tag specifies the type of content contained by the tags.
- Tags can be nested to form hierarchies that further structure the data in a meaningful way.

The only piece of information that might be puzzling is the first line. The first line specifies that the document is formatted using the rules set out by the XML 1.0 standard and that the document is encoded using UTF-8.

This example is straightforward. However, you might ask yourself, "What if Erwin has more than one phone number?" To further structure the data, an XML document can contain any number of the same type of element and can augment the elements with attributes that distinguish the elements, as shown in Listing 3.2.

Listing 3.2 An Expanded XML Example

```
<?xml version="1.0" encoding="UTF-8"?>
<Person>
  <Name>Erwin van der Koogh</Name>
  <Address>12 Lower Hatch Street</Address>
  <City>Dublin</City>
  <Country>Ireland</Country>
  <Phone Type="Home">555-5555</Phone>
  <Phone Type="Work">555-1234</Phone>
</Person>F
```

The addition of the `Type` attribute to the `Phone` element tells you that `555-5555` corresponds to Erwin's home phone number and that `555-1234` corresponds to his work phone number.

More formal XML documents might use DTDs to define the following:

- Which tags are valid for a document
- How many times a specific tag might occur

- The order of the tags
- Required and optional attributes
- Default attribute values

When an XML document implements the rules specified by a DTD, the XML document is said to be valid. When an XML document doesn't use a DTD but otherwise follows the rules of XML, it is said to be well formed. For simple applications of XML, it is usually enough that documents are well formed, eliminating the overhead required to check that a document complies with a DTD.

That, in a nutshell, is about all the XML you need to know or understand to comprehend the XML used by JXTA. Although XML supports many other wonderful capabilities, understanding these capabilities isn't necessary to understand JXTA's use of XML. For more information on XML, see Appendix B, "Online Resources," for the location of the XML standard and other XML resources.

JXTA Advantages and Disadvantages

JXTA provides a far more abstract language for peer communication than previous P2P protocols, enabling a wider variety of services, devices, and network transports to be used in P2P networks. The employment of XML provides a standards-based format for structured data that is well understood, well supported, and easily adapted to a variety of transports. XML also has the advantage that it's a human-readable format, making it easy for developers to debug and comprehend. So far, JXTA seems to have done everything right. Well, maybe not.

One important element that JXTA does not attempt to address is how services (other than the core services) are invoked. Several standards exist for defining service invocation, such as the Web Services Description Language (WSDL), but none has been specifically chosen by the JXTA Protocols Specification. JXTA provides a generic framework for exchanging information between peers, so any mechanism, such as WSDL, could potentially be used via JXTA to exchange the information required to invoke services.

Several other arguments arise against the flexibility that the designers of JXTA infused throughout the JXTA Protocols Specification. Although JXTA's use of XML specifies all aspects of P2P communication for any generic P2P application, JXTA might not be suited to a specific standalone P2P application. In an individual application, the network overhead of XML messaging might be more trouble than it's worth, especially if the application developer has no intention of taking advantage of JXTA's capabilities to incorporate other P2P services into the application.

Critics of JXTA point out that the platform's abstraction of the network transport is another potential area of excess. If most P2P applications today rely on the Transport Control Protocol (TCP) to provide a network transport, why does JXTA go to such lengths to avoid tying the protocols to a specific network transport? Why not specify TCP as the assumed network transport and eliminate the overhead?

All these points highlight the need for developers to balance flexibility with performance when implementing their P2P applications. JXTA might not be the best or most efficient solution for implementing a particular P2P application. However, JXTA provides the most well-rounded platform for producing P2P applications that have the flexibility required to grow in the future. The capability to leverage other P2P services and enable widespread development of P2P communities is the core value of the JXTA platform.

How Is JXTA Different from Jini or .NET?

The promise of interconnecting any type of device over any type of network might sound familiar to followers of Sun's Jini technology. Although there are some similar goals, Jini relies exclusively on the Java platform for its functionality, whereas JXTA has no dependence on a particular programming language. Unlike JXTA, Jini uses a centralized server to locate services on the network and relies on Remote Method Invocation (RMI) and object serialization for communication with remote devices. JXTA relies on XML rather than object serialization to exchange structured data and discovers services across all peers on the P2P network.

The Web Services aspects of Microsoft's .NET platform are heavily infused with XML, but the use of XML alone doesn't make them comparable. Fundamentally, JXTA and .NET have completely different purposes, with .NET focusing more on the traditional client/server architecture of service delivery. Although .NET technology could form the

foundation of a P2P application, creating a full P2P solution with .NET would require extra work on the part of the developer. Developing a P2P solution using .NET would require a developer to specify all the core P2P interactions, such as peer discovery. This solution would essentially involve recreating all the mechanisms that are already defined by the JXTA protocols.

Introducing the JXTA Shell

Rather than try to explain JXTA in the abstract, what better way to start to understand JXTA than seeing the technology in action? To do this, the remainder of this chapter guides you through using the JXTA Shell.

The JXTA Shell is a demo application built on top of the JXTA platform that allows users to experiment with the functionality made available through the Java reference implementation of the JXTA protocols. The JXTA Shell provides a UNIX-like command-line interface that allows the user to perform P2P operations by manipulating peers, peer groups, and pipes using simple commands.

Before You Install the JXTA Shell

To make the installation easier, you should already have a Java Run-Time Environment (JRE), version 1.3 or later, on your computer. To test whether you have a JRE already installed, go to the command prompt and type

```
java –version
```

If you have an existing JRE, you will see version information from the runtime of this form:

```
java version "1.3.1_01"
Java(TM) 2 Runtime Environment, Standard Edition (build 1.3.1_01)
Java HotSpot(TM) Client VM (build 1.3.1_01, mixed mode)
```

If you don't see this type of output, or if your version is lower that 1.3, you need to install a version 1.3 or higher JRE.

You can download a version of the JXTA Shell installer for most platforms with a standalone JRE included. However, if you intend to try the example code in this book's later chapters, you should install the Java 2 SDK (which includes a JRE) instead of a standalone JRE. The Java 2 SDK for most major platforms, including Solaris, Linux, and Windows, is available from www.javasoft.com/j2se/.

For developers using the Mac platform, the latest Java environment can be downloaded from www.apple.com/java/ but is available only for the Mac OS X platform.

Obtaining and Installing the JXTA Shell

The JXTA Shell application can be obtained from either the Project JXTA web site as a set of prebuilt binaries or from the Project JXTA source control system as a set of source files.

To avoid the extra work required to build the JXTA Shell from source code, these experiments use the prebuilt JXTA Shell binaries that come with the JXTA demo applications. To download the JXTA demo installer that includes the JXTA Shell binaries, go to download.jxta.org/easyinstall/install.html.

Installing the JXTA demo applications also installs the latest stable build of the JXTA platform, packaged as a set of Java Archive (JAR) files. Unless you're interested in working with the latest experimental (and potentially unstable) version available from the Project JXTA CVS repository, these archives are all that's required to build new JXTA solutions in Java. The latest JXTA build at the time of writing was build 47b, built on January 25, 2002.

The installation procedure is slightly different for each operating system. The following sections describe the installation procedure for various operating systems.

Installing the JXTA Shell for Windows

To install the JXTA demo applications for the Windows platform, follow these steps:

1. Open download.jxta.org/easyinstall/install.html in a web browser.

2. If you already have a version 1.3 or later JRE installed on your machine, download the Windows Without Java VM installer; otherwise, download the Windows Includes Java VM installer.

3. When prompted by your web browser, specify a directory to store the downloaded installer.

4. After the download is complete, open Windows Explorer and go to the folder where you stored the downloaded installer.

5. Run the installer. It should be called either `JXTAInst.exe` or `JXTAInst_VM.exe`, depending on whether you chose the installer that includes the JVM.

6. Click Next to dismiss the Introduction dialog box.

7. The installer displays the License Agreement dialog box. Select the radio button titled I Accept the Terms of License Agreement, and click Next.

8. The installer prompts you to specify where the JXTA demo applications should be installed. Unless you have reason to install them elsewhere, use the default installation directory provided (`C:\Program Files\JXTA_Demo`). Click Install.

9. The installer installs the JXTA demo applications and then displays instructions on how to run the demo applications. Note these instructions before clicking Next to dismiss the launch instructions.

10. Click Done to close the installer.

The demo applications are now installed, and you can safely delete the installer that you downloaded.

Installing the JXTA Shell for Solaris, Linux, and UNIX

To install the JXTA demo applications for the Windows platform, follow these steps:

1. Open download.jxta.org/easyinstall/install.html in a web browser.

2. If you already have a version 1.3 or later JRE installed on your machine, download the Without Java VM installer for your platform; otherwise, download the Includes Java VM installer for your platform. The UNIX platform install does not have a version that includes a standalone JRE, so if you don't already have a JRE, you must download and install one first.

3. When prompted by your web browser, specify a directory to store the downloaded installer.

4. After the download is complete, open a console and go to the folder where you stored the downloaded installer.

5. Run the installer using `sh ./JXTAInst.bin`, replacing `JXTAInst.bin` with the name of the file that you downloaded. It should be called `JXTAInst.bin`, `JXTAInst_Sol_VM.bin`, or `JXTAInst_LNX_VM.bin`, depending on which version you chose to download.

6. Click Next to dismiss the Introduction dialog box.

7. The installer displays the License Agreement dialog box. Select the radio button titled I Accept the Terms of License Agreement, and click Next.

8. The installer prompts you to specify where the JXTA demo applications should be installed. Unless you have reason to install them elsewhere, use the default installation directory provided (`~/JXTA_Demo`). Click Install.

9. The installer installs the JXTA demo applications and then displays instructions on how to run the demo applications. Note these instructions before clicking Next to dismiss the launch instructions.

10. Click Done to close the installer.

The demo applications are now installed, and you can safely delete the installer that you downloaded.

Installing the JXTA Shell for Other Java-Supported Platforms

To install the JXTA demo applications for any other platform that supports Java, you can download a Java-based installer. However, you must already have a JRE installed on your machine. To install the JXTA demo applications for a Java-enabled platforms, follow these steps:

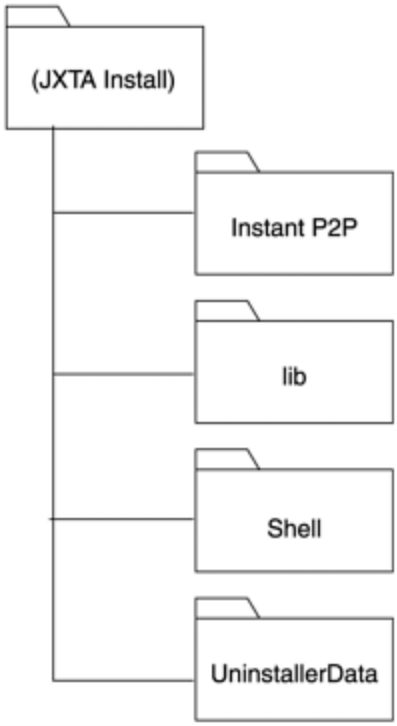
1. Open download.jxta.org/easyinstall/install.html in a web browser.
2. Download the Other Java-Enabled Platforms version of the JXTA Shell installer.
3. When prompted by your web browser, specify a directory to store the downloaded installer.
4. After the download is complete, open a console and go to the folder where you stored the downloaded installer.
5. Run the installer using `java -classpath JXTA_Demo.zip install`.
6. Click Next to dismiss the Introduction dialog box.
7. The installer display the License Agreement dialog box. Select the radio button titled I Accept the Terms of License Agreement, and click Next.
8. The installer prompts you to specify where the JXTA demo applications should be installed. Unless you have reason to install them elsewhere, use the default installation directory provided (usually `C:\Program Files\JXTA_Demo` or `~/JXTA_Demo`). Click Install.
9. The installer installs the JXTA demo applications and then displays instructions on how to run the demo applications. Note these instructions before clicking Next to dismiss the launch instructions.
10. Click Done to close the installer.

The demo applications are now installed, and you can safely delete the installer that you downloaded.

The Installation Directory Structure

When the installation is complete, the directory structure that's shown in Figure 3.3 appears.

Figure 3.3. The installation directory structure.



(`JXTA Install`) is the installation directory that you specified to the installer. The `lib` subdirectory contains the JARs for the JXTA platform and the demo applications, and the `Shell` subdirectory contains the executable to start the Shell application. After the Shell is executed, the `Shell` subdirectory also holds a cache of configuration information and discovered peers and resources.

The `InstantP2P` directory contains another demo application that you do not use here. The `UninstallerData` directory contains the executable required to uninstall the JXTA demo applications.

Running the JXTA Shell

To start the JXTA Shell, follow the instructions provided at the end of the installation process.

On Windows, start the application by clicking Start, Programs, JXTA, JXTA Shell.

On other platforms, execute the script provided by the installer to start the application:

1. Open a command shell.
2. Go to the directory location that you specified for the JXTA Shell during the installation.
3. Go to the `Shell` subdirectory.
4. Execute the `shell.exe` or the `shell.sh script`.

Alternatively, you can invoke the Shell application directly using this command from the `Shell` subdirectory of the JXTA installation:

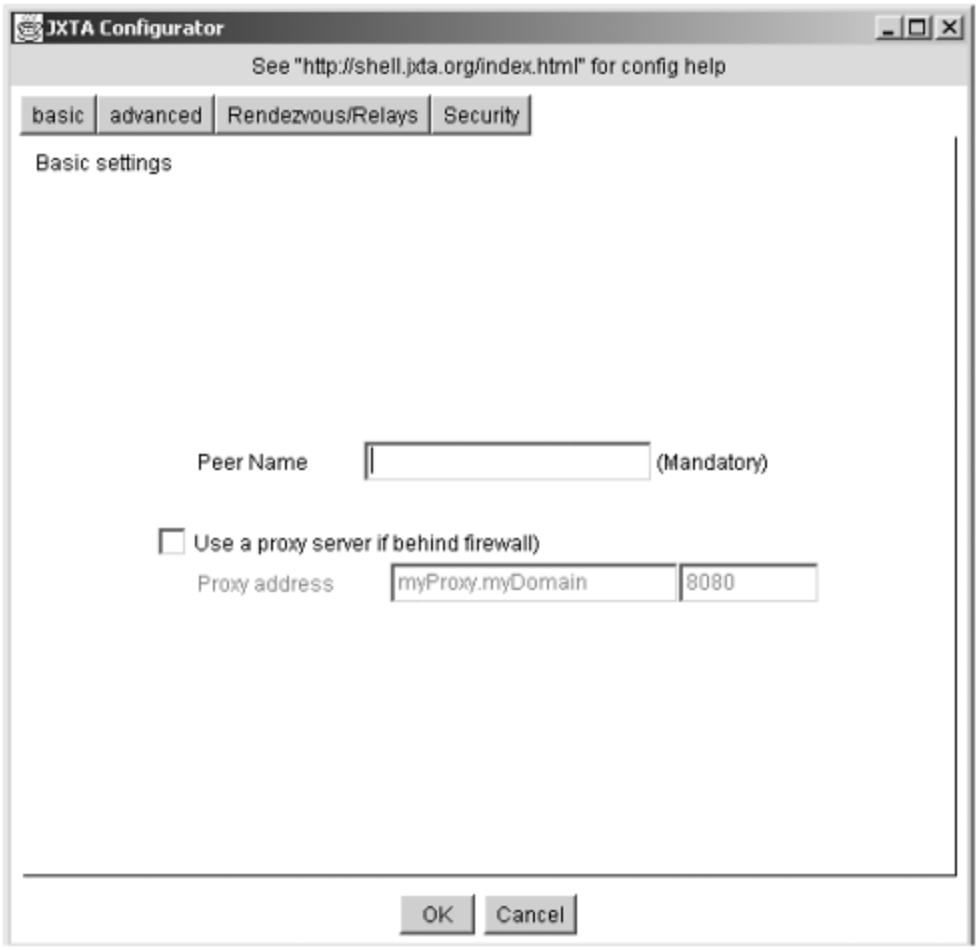
```
C:\Program Files\JXTA_Demo\Shell>java -classpath ..\lib\jxta.jar;
..\lib\jxtashell.jar;..\lib\log4j.jar;..\lib\jxtasecurity.jar;
..\lib\cryptix-asn1.jar;..\lib\cryptix32.jar;..\lib\minimalBC.jar;
..\lib\jxtaptls.jar net.jxta.impl.peergroup.Boot
```

On non-Windows platforms, you need to change the command given to match the directory and environment variable separator characters used by your platform. On Solaris, Linux, and UNIX, use `/` instead of `\`, and `:` instead of `;`.

Configuring the Shell

The first time you execute the application, you are presented with a screen requesting configuration information, as shown in Figure 3.4.

Figure 3.4. The basic configuration screen.



The user interface that appears, called the Configurator, is used by the reference implementation to configure the JXTA platform before starting the JXTA platform. To configure the JXTA platform using the Configurator, follow these steps:

1. Enter a name for your peer in the Peer Name text field.

2. Go to the Security tab.

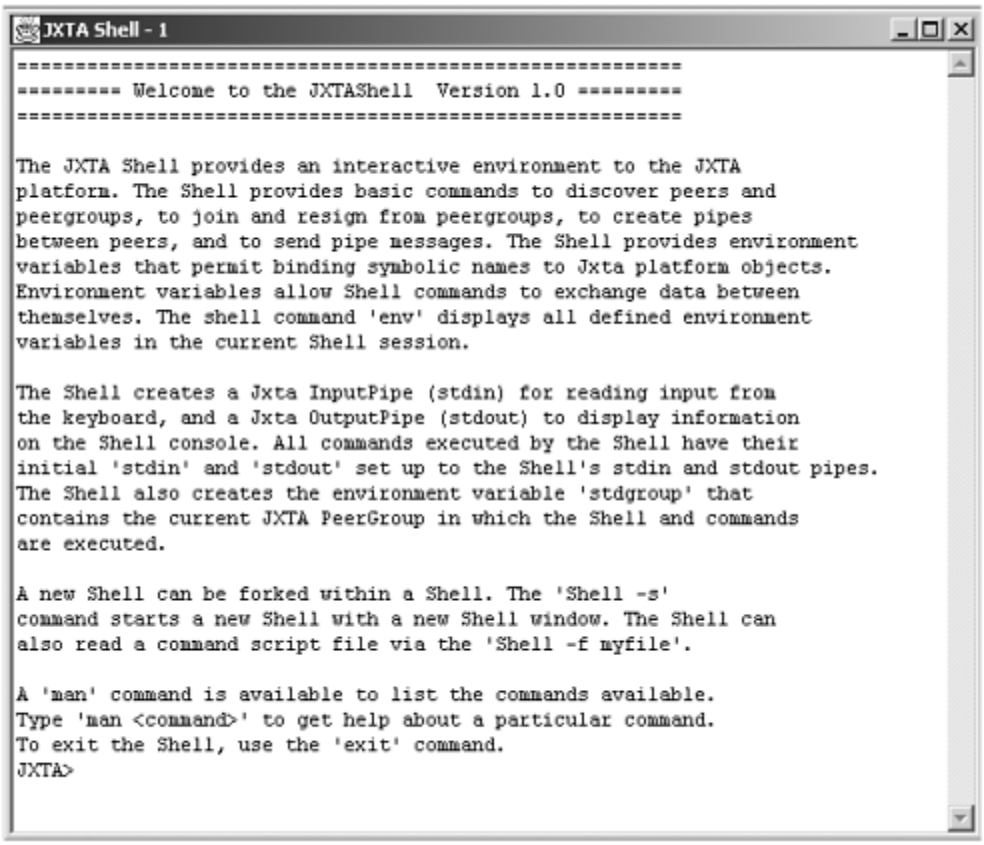
3. Enter a username in the Secure Username text field.

4. Enter a password in the Password text field, and enter the same password in the Verify Password text field. Be sure to note the username and password that you enter because they will be required each time you start the JXTA platform in the future.

5. Click OK.

After you click OK, the JXTA platform starts and connects to the network. The time that it takes to start the JXTA platform varies with speed of your network connection, but it should take less than 30 seconds, at the most. Assuming that you have a simple network configuration, the Shell should start up and display the screen that's shown in Figure 3.5.

Figure 3.5. The JXTA Shell user interface.



To confirm that your client is correctly connected to the network, enter the `rdvstatus` command at the JXTA prompt:

```
JXTA>rdvstatus
```

If the Shell is correctly configured and managed to locate a rendezvous server, the `rdvstatus` command returns a similar result to the one given in Listing 3.3.

Listing 3.3 Results of the *rdvstatus* Command

```
Rendezvous Connection Status:


_______________________________


Is Rendezvous : [false]

Rendezvous Connections :

  Rendezvous name: JXTA.ORG 237
  Rendezvous name: JXTA.ORG 235
  Rendezvous name: ensd_1

Rendezvous Disconnections :

  [None]
```

This output shows that the Shell has correctly connected to three rendezvous peers, named `JXTA.ORG 237`, `JXTA.ORG 235`, and `ensd_1`. If you receive this response, your Shell peer is correctly configured and connected to the network; if you don't receive this response, see the next section to troubleshoot your configuration.

Troubleshooting Your Peer's Configuration

Listing 3.4 shows the output of the `rdvstatus` command when the client has failed to locate any rendezvous peers and cannot locate other peers.

Listing 3.4 No Visible Rendezvous Peers

```
Rendezvous Connection Status:


_______________________________


Is Rendezvous : [False]
```

```
Rendezvous Connections :
[None]

Rendezvous Disconnections :

[None]
```

In some cases, it might take a few moments to see the rendezvous peers due to network latency. Wait a few moments before running `rdvstatus` again to see if the problem is simply high network latency. If the `rdvstatus` still shows no rendezvous peers, try using this command:

```
JXTA>peers -r
```

Wait a few moments and try the `rdvstatus` command again. If `rdvstatus` still fails to show any rendezvous peers, several possible reasons exist:

- No rendezvous peers are available.
- Your firewall configuration is preventing you from communicating with a rendezvous peer.
- You're not connected to a network.

If you aren't connected to a network, you can still use the Shell to experiment with the JXTA platform by following the instructions in the later section, "Using the JXTA Shell Without a Network Connection."

Finding Available Rendezvous Peers

First, confirm that rendezvous peers are available on the network:

1. Force the Shell to display the configuration screen the next time you start the Shell by typing the following from within the Shell:

2.

```
JXTA>peerconfig
```

If you don't invoke this command before exiting the Shell, the Shell simply uses cached configuration information the next time it starts, with the same results. The `peerconfig` command will return this:

```
peerconfig: Please exit and restart the jxta shell to
reconfigure !!!!!
```

2. Follow the instructions and exit the shell by using the following command:

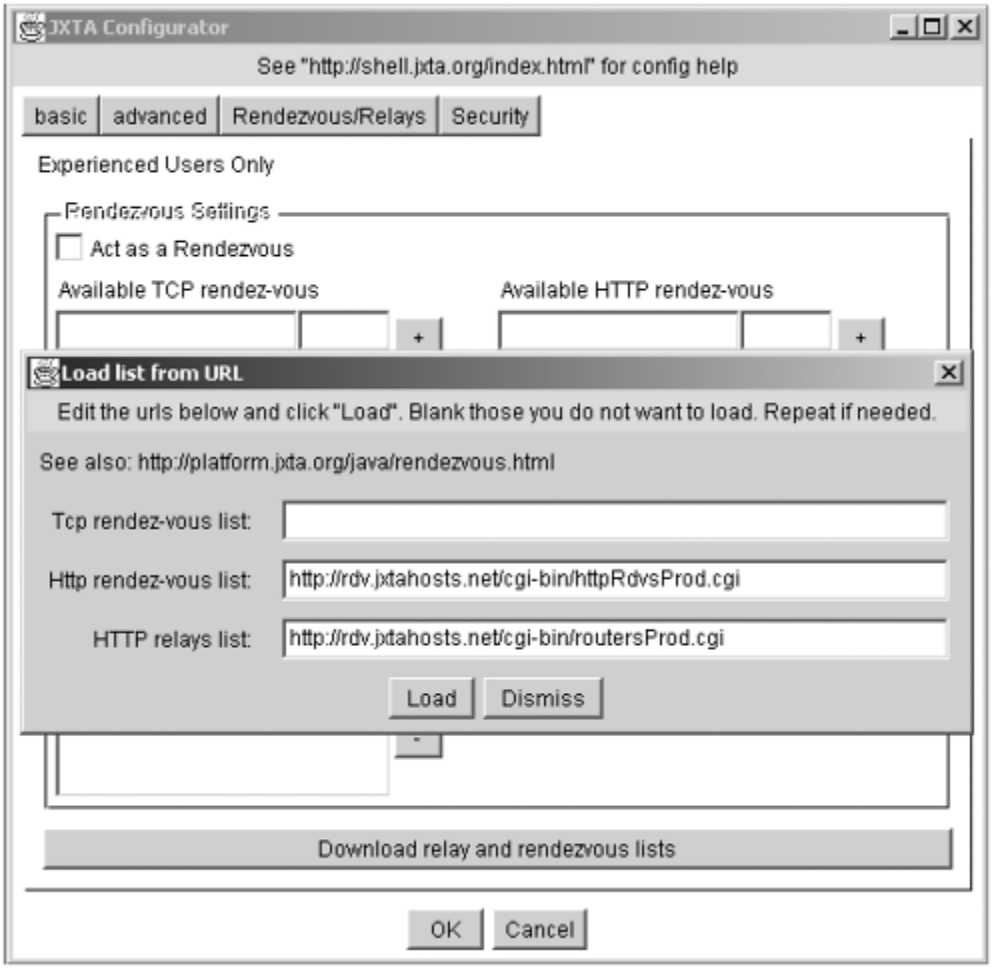
3.

```
JXTA>exit
```

3. Restart the Shell application the same way you started it the first time. This time you are prompted to enter only the username and password that you entered the first time in the Configurator. Enter the username and password, and hit Enter.

4. When the configuration screen appears this time, go to the Rendezvous/ Relays tab and click Download Relay and Rendezvous Lists. The Load list from URL dialog box appears. (See Figure 3.6.)

Figure 3.6. The Download Rendezvous/Router List dialog box.



To find rendezvous peers to use for peer discovery, the JXTA Shell attempts to download a list of available rendezvous peer IP addresses. This is a convenient mechanism for finding rendezvous peers, although you could just as easily enter the IP address and port of a rendezvous peer manually in the Rendezvous Settings section of Rendezvous/Router tab.

Using a web browser, go to the location shown in the Http rendez-vous list text field—by default, this value is as follows:

```
http://rdv.jxtahosts.net/cgi-bin/httpRdvsProd.cgi
```

This site returns a list of the production rendezvous peers run by Project JXTA. These peers are running the latest stable release of the JXTA platform, which should be the same as the platform version provided with the JXTA demo application installer. If the page

returned is empty, no known production rendezvous peers are available from Project JXTA. Most likely, this is only a temporary situation occurring during an update to the rendezvous peer software. Try again later, or see the next section, "Using the JXTA Shell Without a Network Connection" for further instructions.

If the URL returns a list of rendezvous peers, you should test to make sure that at least one rendezvous peer in the list is operating at the specified IP address and port. To do this, you can use a web browser to request an acknowledgement from the rendezvous peer. For example, if the rendezvous peer is located at IP address 63.81.220.34 and is listening on port 9700, pointing a web browser to `http://63.81.220.34:9700/ping/` should return a blank web page.You can view the source of the web page to confirm that the result is a web page, not an error page.

If you have found a working rendezvous peer, the problem is mostly likely due to the configuration of a firewall between your machine and the outside network. Go to the Basic tab, check the Use a Proxy Server option, and enter the location of a HTTP proxy on your local network.You can obtain the location of your network's HTTP proxy by copying the proxy settings from your web browser or talking to your network administrator. The Shell should now show rendezvous peers when you start the application and run the `rdvstatus` command.

Using the JXTA Shell Without a Network Connection

If, for some reason, you don't have network access, you can still explore JXTA using the Shell and the experiments in the rest of this book. The Shell application is a peer like any other on a JXTA P2P network, so all the standard commands to manipulate peers, peer groups, and pipes will work exactly the same, independent of the location of the peer. However, you will be able to see, manipulate, and communicate only with your own peer.

If you want to experiment with the JXTA Shell in a more realistic environment, you can run two instances of the Shell on the same machine, using one of the Shell instances as a rendezvous peer. Due to the way the Java reference implementation of the JXTA platform implements its cache of configuration information, you need to make a copy of the Shell directory to prevent clashes between the instances of the Shell:

1. Force the Shell to display the configuration dialog box the next time it starts using the `peerconfig` command from within the Shell.

2. Exit the Shell using `exit`.

3. Make a copy of the `Shell` subdirectory (located underneath the JXTA installation directory) called `Shell2` at the same level as the `Shell` subdirectory.

Before attempting to configure each Shell, you should know your machine's local IP address.

On Windows, follow these steps:

1. Open a command prompt.

2. Invoke the `ipconfig` command.

3. Note the IP address specified in the output from `ipconfig`.

You should also ensure that you can ping your own IP address because some internal networks might not allow you to see your own IP address. To check if you can see your own IP address, follow these steps:

1. Open a command prompt.

2. Invoke the `ping` command.

3. Ensure that the response doesn't indicate that the destination host is unreachable.

If you cannot ping the IP address returned by `ipconfig`, you should use the localhost IP address 127.0.0.1 instead of the IP address returned by `ipconfig`. On other operating systems, consult your operating system's help system to learn how to determine your machine's IP address and ping an IP address.

To start one Shell as a rendezvous peer, open a command prompt and follow these steps:

1. Go to the `Shell` subdirectory.

2. Start the Shell using the executable or script in the directory directly (`shell.exe` or `shell.sh`) or manually using the `java` command.

3. Enter a name for the peer.

4. Go to the Rendezvous/Relays tab.

5. Remove each TCP and HTTP rendezvous server and each HTTP relay server.

6. Deselect Use a Relay in the HTTP Relay Settings section.

7. Select Act as a Rendezvous.

8. Go to the Advanced tab.

9. Deselect Enabled from the HTTP Settings section.

10. Select Enabled and Manual from the TCP Settings section.

11. Select Always Manual, and note the IP address and port number (default 9701) that has been automatically set. If no IP address has been set, enter the IP address that you obtained from your operating system.

12. Click OK.

13. Enter your username and password when prompted, and hit Enter.

To start a second Shell to act as a simple peer using the rendezvous peer that you just created, open a second command prompt and do the following:

1. Go the `Shell2` subdirectory that you created.

2. Remove the `pse` subdirectory. This directory contains the personal security settings protected by the password entered in the Configurator.

3. Remove the `PlatformConfig` file. This file contains configuration information for your peer, and it must be removed to prevent the second instance from reusing the peer's unique ID.

4. Start the Shell using the executable or script in the directory directly (`shell.exe` or `shell.sh`) or manually using the `java` command.

5. Enter a name for the peer, preferably one that is different from the one you used for the rendezvous peer.

6. Go to the Rendezvous/Relays tab.

7. Remove each TCP and HTTP rendezvous server and each HTTP relay server.

8. Deselect Use a Relay in the HTTP Relay Settings section.

9. Enter the IP address and port that you noted in the first shell as a TCP Rendezvous, and add it to the list using the + button.

10. Go to the Advanced tab.

11. Select Enabled and Manual from the TCP Settings section.

12. Select Always Manual, and enter your IP address and a different port number (say, 9702).

13. Deselect Enabled from the HTTP Settings section.

14. Go the Security tab.

15. Enter a username and password.

16. Click OK.

You now have a simple peer configured to use the first instance of the Shell as a rendezvous peer, and you can conduct P2P communication between the two peers as normal.

Navigating the JXTA Shell

The JXTA Shell presents a simple command-line user interface similar to UNIX's interface. Simple text commands are entered at the JXTA prompt:

```
JXTA>
```

Like most UNIX shells, the Shell is case-sensitive and maintains a history of previously issued commands. At any time, you can see a complete list of the previously issued commands by using the `history` command:

```
JXTA>history
  0 man
  1 history
```

At any time, you can scroll through the commands using the up and down arrow keys, invoking previous commands without retyping the command.

Learning About Shell Commands

The JXTA Shell resembles a UNIX shell in many ways, and several of the commands are available within the Shell. To learn what commands are available from the Shell, you can use the `man` command by itself to print a list of all available commands, shown in Table 3.1:

```
JXTA>man
```

Table 3.1. Built-In Shell Commands

Command	Description
`cat`	Concatenates and displays a Shell object
`chpgrp`	Changes the current peer group
`clear`	Clears the shell's screen
`env`	Displays environment variable
`exit`	Exits the Shell
`exportfile`	Exports to an external file
`get`	Gets data from a pipe message
`grep`	Searches for matching patterns
`groups`	Discovers peer groups
`help`	Gives instructions on where to find help
`history`	Shows the history of Shell commands executed
`importfile`	Imports an external file
`instjar`	Installs JAR files containing additional Shell commands
`join`	Joins a peer group
`leave`	Leaves a peer group
`man`	Online help command that displays information about a specific Shell command
`mkadv`	Makes an advertisement

`mkmsg`	Makes a pipe message
`mkpgrp`	Creates a new peer group
`mkpipe`	Creates a pipe
`more`	Pages through a Shell object
`peerconfig`	Forces reconfiguration the next time the Shell is started
`peerinfo`	Gets information about peers
`peers`	Discovers peers
`put`	Puts data into a pipe message
`rdvserver`	Runs the peer as a standalone rendezvous server
`rdvstatus`	Displays information about rendezvous
`recv`	Receives a message from a pipe
`search`	Discovers JXTA advertisements
`send`	Sends a message into a pipe
`set`	Sets an environment variable
`setenv`	Sets an environment variable
`sftp`	Sends a file to another peer
`share`	Shares an advertisement
`Shell`	Forks a new JXTA Shell command interpreter
`Sql`	Issues an SQL command (not implemented)
`Sqlshell`	Acts as the JXTA SQL Shell command interpreter
`Talk`	Talks to another peer
`Uninstjar`	Uninstalls JAR files previously installed with `instjar`
`Version`	Returns the Shell version information
`wc`	Counts the number of lines, words, and characters in an object
`who`	Displays credential information
`whoami`	Displays information about a peer or a peer group

The `man` command also enables you to learn about the purpose and options for various commands available within the Shell. For example, to find out more about the `rdvstatus` command, use this command:

```
JXTA>man rdvstatus
```

This pulls up the usage information for the `rdvstatus` command, as shown in Listing 3.5.

Listing 3.5 Usage Information for *rdvstatus*

```
NAME
      rdvstatus - display information about rendezvous

SYNOPSIS
      rdvstatus [-v]
```

```
        [-v]    print verbose information

DESCRIPTION

rdvstatus displays information about the peer
rendezvous. The command shows how many rendezvous peers
the peer is connected to.

OPTIONS
        -v    print verbose information
EXAMPLE
    JXTA>rdvstatus

SEE ALSO
    whoami peers
```

Environment Variables

The Shell provides environment variables to store pieces of information in the Shell for later use.You can see the defined environment variables using the `env` command, as shown in Listing 3.6.

Listing 3.6 The Shell Environment Variables

```
JXTA>env
stdin = Default InputPipe (class net.jxta.impl.shell.ShellInputPipe)
SHELL = Root Shell (class net.jxta.impl.shell.bin.Shell.Shell)
History = History (class net.jxta.impl.shell.bin.history.HistoryQueue)
parentShell = Root Shell (class net.jxta.impl.shell.bin.Shell.Shell)
Shell = Root Shell (class net.jxta.impl.shell.bin.Shell.Shell)
stdout = Default OutputPipe (class
net.jxta.impl.pipe.NonBlockingOutputPipe)
consout = Default Console OutputPipe (class
net.jxta.impl.shell.ShellOutputPipe)
consin = Default Console InputPipe (class
net.jxta.impl.shell.ShellInputPipe)
```

```
stdgroup = Default Peer Group (class net.jxta.impl.peergroup.StdPeerGroup)
```

These environment variables are set by default to handle the input, output, and basic functionality of the Shell. More variables can be defined by the output of commands, each corresponding to an object, data, or cached advertisement accessible within the Shell's environment.

Importing and Exporting Environment Variables

In addition to working with environment variables within the Shell, environment variables can be imported and exported using the `importfile` and `exportfile` commands. The commands enable you to import XML or plain text files into Shell environment variables. By default, the working directory for these commands is set to the directory where you executed the Shell, usually the Shell subdirectory of the JXTA installation directory.

To demonstrate the `importfile` and `exportfile` commands, follow these steps:

1. Create a text file called `input.txt` containing some text in the Shell subdirectory under the JXTA installation directory.

2. Import the text of the file into an environment variable called `test` using `importfile -f input.txt test`.

You should see a new environment variable named `test` in the list of variables returned by the `env` command. Rather than trying to find a variable in the output of `env`, you can use the `cat` command to view the contents of the `test` variable, as shown in Listing 3.7:

```
JXTA>cat test
```

Listing 3.7 The Imported Environment Variable

```
<?xml version="1.0"?>

<ShellDoc>
      <Item>
           This is some test text.
      </Item>
```

```
</ShellDoc>
```

The `cat` command knows how to render several types of environment variables to the standard output of the Shell, including the XML document produced by importing `input.txt` with the `importfile` command. The `test` environment variable can now be exported to a file called `output.txt` using this command:

```
JXTA>exportfile –f output.txt test
```

A file called `output.txt` containing the contents of the `test` variable appears in the Shell subdirectory of the JXTA installation.

Although the usefulness of this functionality might seem trivial now, remember that all functionality in JXTA is expressed in terms of XML-based advertisements. As you'll see, having the capability to manipulate environment variables is central to the power of the JXTA Shell as a tool for experimenting with the JXTA platform.

Combining Commands

In a manner similar to UNIX, the JXTA Shell allows users to string together simple commands to perform complex functionality. The | operator allows the output of a command on the left of the operator to be used as input into the command on the right.

Consider a simple example: The output of the `man` command is a bit too long to read without scrolling. The `more` command breaks a piece of input text into screen-size chunks.You can combine these two commands by typing the following:

```
JXTA>man | more
```

This pipes the output of the `man` command as input into the `more` command, allowing you to view the `man` output in screen-size chunks that you can move between by hitting the Enter key. Similarly, you could count the number of characters in the `man` output using this command:

```
JXTA>man | wc –c
```

This pipes the output of the `man` command as input into the `wc` command, which counts the number of characters in the input when the `-c` option is set.

Manipulating Peers

The JXTA Shell provides basic capabilities to discover peers and obtain peer information. Working with a peer involves working with a Peer Advertisement that describes the peer and its services.

Learning About the Local Peer

Before learning about other peers, you need to know a bit about your own local peer:

```
JXTA>whoami
```

The `whoami` command returns the peer information for the local peer run by the JXTA Shell given in Listing 3.8.

Listing 3.8 Results of the *whoami* Shell Command

```
<Peer>MyPeer</Peer>
<Keywords>NetPeerGroup by default</Keywords>
<PeerId>urn:jxta:uuid-59616261646162614A78746150325033855A703D4E614D
B7B54A9BE583FFCD4C03</PeerId>
<TransportAddress>tcp://asterix:9701/</TransportAddress>
<TransportAddress>http://JxtaHttpClientuuid-59616261646162614A787461
50325033855A703D4E614DB7B54A9BE583FFCD4C03/</TransportAddress>
```

This short version of the local peer information shows only the basic peer information. A longer version that displays the whole Peer Advertisement stored in environment variable `peerX` can be viewed using this command:

```
JXTA>cat peerX
```

You can find the environment variable holding your Peer Advertisement by looking for your peer's name in the results of the `peers` command.

I won't go into the details of the Peer Advertisement at this point. I will provide a complete description of the Peer Advertisement when we explore the Peer Discovery Protocol in the next chapter. For now, it's enough to notice some of the information provided by the advertisement:

- A name for the peer
- A unique identifier for the peer
- Services provided by the peer
- Transport endpoint details

The services provided by the peer are called peer services; these are services offered only by the peer. If the peer disconnects from the network, these services are unavailable to other peers.

Finding Peers

Before your peer can request services from a peer, it needs to know the existence of the peer, what services the peer offers, and how to contact the peer on the network. To find peers that your local peer is already aware of, execute the `peers` command given in Listing 3.9.

Listing 3.9 Results of the *peers Shell* Command

```
JXTA>peers
peer0: name = rdv-235
peer1: name = rdv-237
peer2: name = dI_lab1
peer3: name = dI_lab_Tokyo
peer4: name = MyPeer
```

Each Peer Advertisement is made available in the Shell environment via a variable with a name of the form `peerX`, where `X` is an integer. At this point, your peer is aware of only local or cached Peer Advertisements for peers that have already been discovered; no discovery of remote peers has yet been performed. Caching Peer Advertisements reduces

the amount of discovery that a peer might have to perform and can be used by simple peers as well as rendezvous peers to reduce network traffic.

Each entry returned by the `peers` command shows the simple peer name for a peer and the name of an environment variable storing the Peer Advertisement for that peer. In the previous example, the `peer4` environment variable stores the Peer Advertisement for the local peer.You can view the Peer Advertisement using the `cat` command:

```
JXTA>cat peer4
```

To discover other peers on the network, you need to send a peer discovery message using the following:

```
JXTA>peers -r
peer discovery message sent
```

This sends a discovery message immediately to all the rendezvous peers that your peer is aware of on the network. The rendezvous peers forward the request to other rendezvous and simple peers that it is aware of on the network. The rendezvous peers might potentially reply using cached Peer Advertisements to improve the response time and reduce network traffic across the P2P network. The `peers` command returns to the JXTA prompt immediately, and the discovered peers can be viewed using the `peers` command, as shown in Listing 3.10.

Listing 3.10 The Updated List of Discovered Peers

```
JXTA>peers
peer0: name = cajunboy
peer1: name = fds
peer2: name = Rdv-235
peer3: name = domehuhu
peer4: name = MyPeer
peer5: name = Rdv-236
...
```

The results of the peer discovery might not be immediately viewable with the `peers` command. JXTA provides no guarantees about the time required to receive a response to a

discovery message; it is possible that responses might never return. The delay depends on a variety of factors, including the speed of your connection to other peers and the network configuration (firewall, NAT).

Flushing Cached Peer Advertisements

At some point, it might be appropriate to remove the Peer Advertisements from the local cache, eliminating the local peer's knowledge of other peers on the network. To flush the local cache of Peer Advertisements, use this command:

```
JXTA>peers -f
```

The only remaining Peer Advertisement will be that of your own local peer:

```
JXTA>peers
peer0: name = MyPeer
```

To find peers on the network, you need to send another peer discovery message to the network using the `peers -r` command to populate the local cache of Peer Advertisements.

Manipulating Peer Groups

In the same manner that you just managed to discover and manipulate peer information, you can discover and manipulate peer groups. Working with a peer group involves working with a Peer Group Advertisement that describes the peer group and its services.

Learning About the Current Peer Group

The `whoami` command permits you to examine the peer group information for the local peer's current peer group. In the Shell, the peer can manipulate only one peer group at a time. For convenience, this peer group is set as the current peer group in an environment variable called `stdgroup`. To retrieve information about the current peer group, use `whoami -g` to obtain the peer group information in a form similar to this:

```
<PeerGroup>NetPeerGroup</PeerGroup>
<Description>NetPeerGroup by default</Description>
<PeerGroupId>urn:jxta:jxta-NetGroup</PeerGroupId>
```

This peer group information shows that the peer is currently a part of the Net Peer Group. By default, all peers are members of the Net Peer Group, thereby allowing all peers on the network to see and communicate with each other.

The peer group information returned by `whoami -g` is a condensed version of the information provided by the peer group's advertisement. A Peer Group Advertisement also contains information on the set of services that the peer group makes available to its members. These services are called peer group services to distinguish them from peer services. Peer group services can be implemented by several members of a peer group, enabling redundancy. Unlike a peer service, a peer group service remains available as long as one member of the peer group is connected to the P2P network and is providing the service.

Finding Peer Groups

In a similar manner to viewing the known peers on the network, you can view the known peer groups using this command:

```
JXTA>groups
```

As with the `peers` command, only those peer groups that have been discovered in the past and have had their Peer Group Advertisement cached appear in the list when this command is executed in an instance of the Shell. Although all peers belong to the Net Peer Group and this group is always present, the Net Peer Group does not show up in the results from the `groups` command.

To find peer groups available on the P2P network, a peer group discovery request must be made to the network:

```
JXTA>groups -r
group discovery message sent
```

Using the `groups` command again returns a list of groups discovered on the network:

```
JXTA>groups
group0: name = SomeGroup
group1: name = AnotherGroup
...
```

As with peer discovery, the response to a group discovery message might not be immediate, if a response is obtained at all. Each of the cached Peer Group Advertisements is available in the environment as a variable with a name of the form `groupX`, where `X` is an integer. The contents of the environment variable can be viewed using the `cat` command:

```
JXTA>cat group0
```

This command displays the full Peer Group Advertisement instead of the condensed version returned by `whoami -g`.

Creating a Peer Group

A new peer group can be created from within the JXTA Shell in two ways: by cloning the Net Peer Group Peer Group Advertisement or by creating a new Peer Group Advertisement from scratch.

Cloning The Net Peer Group

To create a new peer group, use the `mkpgrp` command and provide a name for your peer group:

```
JXTA>mkpgrp MyGroup
```

Used this way, the `mkpgrp` command makes a new peer group by cloning the existing Net Peer Group peer group.

Creating a New Peer Group Advertisement

Instead of cloning the existing Net Peer Group, you can create a new Peer Group Advertisement with a given name using this command:

```
JXTA>MyGroupAdvertisement = mkadv -g <name>
```

This form of the `mkadv` command creates a new Peer Group Advertisement by cloning the current peer group. If you haven't yet joined any groups, the current peer group is the Net Peer Group, and the result is identical to using the `mkpgrp` command. Alternatively, you can import a saved Peer Group Advertisement from a text file and use it to create the advertisement:

```
JXTA>importfile -f advertisement.txt MyDocument
JXTA>MyAdvertisement = mkadv -g -d MyDocument
JXTA>mkpgrp -d MyAdvertisement MyGroup
```

This set of commands imports a file called `advertisement.txt`, creates a Peer Group Advertisement out of its contents, and uses them to create a new peer group called `MyGroup`.

Note

Currently, the Shell ignores the `MyGroup` name for the peer group and uses the name from the Peer Group Advertisement; this is a known bug with the current Shell implementation.

Joining a Peer Group

When your peer is aware of a peer group, either by creating one or by performing peer group discovery, you must join the group before any communication as a part of that peer group can occur. To join a group whose Peer Group Advertisement is stored in an environment variable called `group1`, use the `join -d` command:

```
JXTA>join -d group1
```

The `join` command prompts you for an identity that you want to use on this group:

```
Enter the identity you want to use when joining this peergroup (nobody)
```

```
Identity:
```

Identities assign credentials to users for accessing peer resources. The peer group's Membership service is responsible for defining accepted identities and authenticating peers that want to join a group.

The `join -d` command sets the current peer group in the environment to the most recently joined peer group. Issuing the `join` command again lists the current known groups and their status:

```
JXTA>join
 Unjoined Group : AnotherGroup
 Joined Group   : MyGroup       (current)
 Unjoined Group : SomeGroup
```

If you make another group called `MyGroup2` and join it, the current peer group changes to reflect `MyGroup2` as the current peer group:

```
JXTA>join
 Unjoined Group : AnotherGroup
 Joined Group   : MyGroup
 Joined Group   : MyGroup2       (current)
 Unjoined Group : SomeGroup
```

To move between peer groups, change the current shell peer group by issuing the `chpgrp` command, as shown in Listing 3.11.

Listing 3.11 Changing the Current Peer Group

```
JXTA>chpgrp MyGroup
JXTA>join
 Unjoined Group : AnotherGroup
 Joined Group   : MyGroup       (current)
 Joined Group   : MyGroup2
 Unjoined Group : SomeGroup
```

If you decide to leave a peer group, issue the `leave` command; your peer will leave the current peer group, as shown in Listing 3.12.

Listing 3.12 Result of Leaving a Group

```
JXTA>leave
JXTA>join
 Unjoined Group : AnotherGroup
 Unjoined Group : MyGroup
 Joined Group   : MyGroup2
 Unjoined Group : SomeGroup
```

After you leave a peer group, the current peer group is set to the Net Peer Group.You must issue a `chpgrp` command to set the current peer group again.

Flushing Cached Peer Group Advertisements

Just as it might be appropriate to remove the Peer Group Advertisements from the local cache, it might also be appropriate to remove peer group advertisements from the local cache. To flush the local cache of Peer Group Advertise-ments, thereby eliminating the local peer's knowledge of peer groups on the network, use this command:

```
JXTA>groups -f
```

To join a peer group on the network, you need to send another peer group discovery message to the network using the `groups -r` command to populate the local cache of Peer Group Advertisements.

Manipulating Pipes

Pipes provide the basic mechanism for peers to share information with each other. Pipes and pipe endpoints are abstractions of the underlying network-transport mechanism responsible for providing network connectivity. Communicating with other peers involves discovering pipes and endpoints, binding to a pipe, and sending and receiving messages through the pipe.

Creating Pipes

To create a pipe, you must first create a Pipe Advertisement:

```
JXTA>MyPipeAdvertisement = mkadv -p
```

Using the `cat` command, you can view the newly created Pipe Advertisement, as shown in Listing 3.13.

Listing 3.13 Viewing the New Pipe Advertisement

```
JXTA>cat MyPipeAdvertisement
<?xml version="1.0"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
     <Id>
          urn:jxta:uuid-59616261646162614E504720503250339C0C74ADD709
4CEC90EC9D4471DFED5304
     </Id>
     <Type>JxtaUnicast</Type>
</jxta:PipeAdvertisement>
```

When a peer has a Pipe Advertisement, defining a pipe from the Pipe Advertisement is as simple as using these commands:

```
JXTA>MyInputPipe = mkpipe -i MyPipeAdvertisement
JXTA>MyOutputPipe = mkpipe -o MyPipeAdvertisement
```

This defines an input and an output pipe from the advertisement stored in the `MyPipeAdvertisement` environment variable.

Creating a Message

Communication between an input and an output pipe relies on the capability to form a message object to exchange. If you import a text file into the Shell, you can package it inside a message:

```
JXTA>importfile -f test.txt SomeData
JXTA>MyMessage = mkmsg
JXTA>put MyMessage MyData SomeData
```

The last line places the contents of the `SomeData` variable inside an element called `MyData`, as shown in Listing 3.14.

Listing 3.14 The Newly Created Message

```
JXTA>cat MyMessage
Tag: MyData
Body:
<?xml version="1.0"?>

<ShellDoc>
      <Item>
           This is some test text.
      </Item>
</ShellDoc>
```

Sending and Receiving Messages

To demonstrate how simple it is to send a message using a pipe, you're going to send a message from the peer to itself. To send the message from the peer, first define the input and output pipes:

```
JXTA>MyPipeAdvertisement = mkadv -p
JXTA>MyInputPipe = mkpipe -i MyPipeAdvertisement
JXTA>MyOutputPipe = mkpipe -o MyPipeAdvertisement
```

Next, import a file that will form the body of the message:

```
JXTA>importfile -f test.txt SomeData
JXTA>MyMessage = mkmsg
JXTA>put MyMessage MyData SomeData
```

Now send the message:

```
JXTA>send MyOutputPipe MyMessage
```

To receive the message from the pipe, use this command:

```
JXTA>ReceivedMessage = recv -t 5000 MyInputPipe
```

This command attempts to receive a message from the `MyInputPipe` input pipe and store it in the `ReceivedMessage` variable. The command attempts to receive a message for only five seconds before timing out.

If the attempt to receive a message is successful, the command returns the following:

```
recv has received a message
```

The data can be extracted from the received message, as shown in Listing 3.15.

Listing 3.15 Viewing the Received Message Data

```
JXTA>NewData = get ReceivedMessage MyData
JXTA>cat NewData
<?xml version="1.0"?>

<ShellDoc>
      <Item>
           This is some test text.
      </Item>
</ShellDoc>
```

If no message is available to be received, the Shell reports the following:

```
recv has not received any message
```

The Shell recognizes whether a pipe is not the appropriate type required to send or receive a message. Attempting to send using an input pipe instead of an output pipe results in an error:

```
JXTA>send MyInputPipe MyMessage
send: MyInputPipe is not an OutputPipe
java.lang.ClassCastException: net.jxta.impl.pipe.InputPipeImpl
```

Similarly, attempting to receive using an output pipe instead of an input pipe results in an error:

```
JXTA>inputMessage = recv -t 5000 outputPipe
wait: outputPipe is not an InputPipe
java.lang.ClassCastException: net.jxta.impl.pipe.NonBlockingOutputPipe
```

Talking to Other Peers

The `talk` command is a simple application written on top of the JXTA Shell that allows you to talk to other peers. To do this, first create a talk advertisement for a specific username:

```
JXTA>talk –register myusername
```

This has to be done only once as the platform caches the advertisement. Next, start a talk listener daemon using this command:

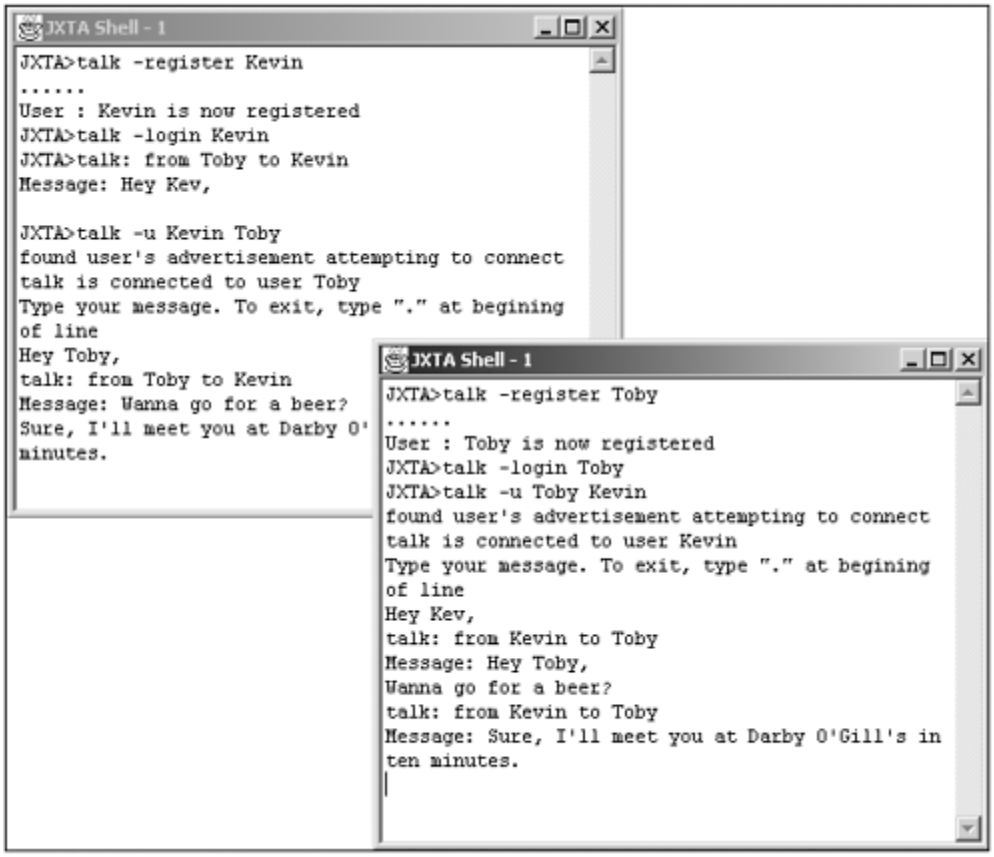
```
JXTA>talk –login myusername
```

After this, you can talk to another user:

```
JXTA>talk –u myusername myfriendsusername
```

This allows you to enter text messages that will be sent to the other talk user `myfriendsusername`, as shown in Figure 3.7.You can even send a text message to yourself using this command:

Figure 3.7. Using `talk` between two shell instances.




```
JXTA>talk -u myusername myusername
```

When you're done talking for the session, use this command to shut down the `talk` daemon:

```
JXTA>talk -logout myusername
```

Extending the Shell Functionality

The JXTA Shell is designed to be more than just a toy to explore the basic building blocks of P2P technology. The Shell is designed to allow developers to extend its functionality easily and incorporate new commands. All the core commands that you've used so far are

invoked dynamically, and any new commands that a developer creates will be invoked the same way.

A developer needs to follow a few simple rules to create a new command for the Shell. To work in the Shell properly, a new command must do the following:

- Extend the `net.jxta.impl.shell.ShellApp` class
- Implement the `startApp` and `stopApp` methods
- Be part of a subpackage of `net.jxta.impl.shell.bin`
- Exist in a subpackage of the same name as the command
- Be in a class of the same name as the command

A Simple Shell Command

Following these simple rules, you'll now write a simple command to print the name of the peer. Listing 3.16 creates a command called `helloworld`.

Listing 3.16 The helloworld Shell Command (*helloworld.java*)

```
package net.jxta.impl.shell.bin.helloworld;

import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;

import net.jxta.peergroup.PeerGroup;

/**
 * A simple example command for the JXTA Shell.
 */
public class helloworld extends ShellApp
{
    /**
```

```
 * The shell environment.
 */
private ShellEnv theEnvironment;

/**
 * Invoked by the Shell to starts the command.
 *
 * @param   args a set of arguments passed to the command.
 * @return  a status code indicating the success or failure
 *          of the command.
 */
public int startApp(String[] args)
{
    println("Starting command...");

    // Get the shell's environment.
    theEnvironment = getEnv();

    // Use the environment to obtain the current peer group.
    ShellObject theShellObject = theEnvironment.get("stdgroup");
    PeerGroup aPeerGroup = (PeerGroup) theShellObject.getObject();

    // Check to see if there were any command arguments.
    if ((args == null) || (args.length == 0))
    {
        // Print the peer name to the console.
        println("My peer name is " + aPeerGroup.getPeerName());
    }
    else
    {
        println("This command doesn't support arguments.");

        // Return the 'parameter error' status code.
        return ShellApp.appParamError;
    }
```

```
        // Return the 'no error' status code.
        return ShellApp.appNoError;
    }

    /**
     * Invoked by the Shell to stop the command.
     */
    public void stopApp()
    {
        // Do nothing.
    }
}
```

As demanded by the rules of the Shell, the `helloworld` class is a part of the `net.jxta.impl.shell.bin.helloworld` package and implements the `startApp` and `stopApp` methods. In this simple example, the command retrieves an object representing the current peer group using the `stdgroup` environment variable:

```
ShellObject theShellObject =
    theEnvironment.get("stdgroup");
```

The `ShellEnv` object is the same store of environment objects that you've been working with from inside the Shell throughout this chapter. The `PeerGroup` object is retrieved from the wrapper `ShellObject` returned by `ShellEnv`:

```
PeerGroup aPeerGroup =
    (PeerGroup) theShellObject.getObject();
```

Finally, the name of the peer in the peer group is printed to the console using the Shell's standard output:

```
println("My peer name is " +
    aPeerGroup.getPeerName());
```

To make this command work with the Shell, compile the `helloworld.java` source from the command line. To make life easier, place the source code in the `Shell` subdirectory of the JXTA demo installation and compile it using the following:

```
javac –d . –classpath ..\lib\jxta.jar;..\lib\jxtashell.jar helloworld.java
```

Now execute the Shell, making sure to include the current directory in the classpath:

```
java -classpath .;..\lib\jxta.jar;..\lib\jxtashell.jar;..\lib\cms.jar;
..\lib\cmsshell.jar;..\lib\log4j.jar;..\lib\beepcore.jar;
..\lib\cryptix32.jar;..\lib\cryptix-asn1.jar;..\lib\jxtaptls.jar;
..\lib\jxtasecurity.jar;. net.jxta.impl.peergroup.Boot
```

The Shell starts up as usual, and you can now try your new command:

```
JXTA>helloworld
Starting command...
My peer name is MyPeer
```

Congratulations, you just created your first solution using JXTA! Although this example doesn't do much, it demonstrates how simple it is to build on the JXTA platform to incorporate new functionality.

Summary

This chapter provided a crash course on using the JXTA Shell. Most of the details of JXTA, its protocols, and the Java reference implementation are revealed in the following chapters. In the next chapter, you start examining the JXTA platform in detail by looking at the Peer Discovery Protocol and its components.Your familiarity with the JXTA Shell will come in handy by providing a framework for the examples, thereby reducing the amount of coding required and allowing the examples to focus on the particulars of peer discovery.

Part II: JXTA Protocols

Chapter 4. The Peer Discovery Protocol

As described in Chapter 2, "P2P concepts," advertisements are the basic unit of data exchanged between peers to provide information on available services, peers, peer groups, pipes, and endpoints. With advertisements, the problem of finding peers and all their different types of resources can be reduced to a problem of finding advertisements describing those resources.

The Peer Discovery Protocol (PDP) defines a protocol for requesting advertisements from other peers and responding to other peers' requests for advertisements. This chapter describes the format of the messages of the PDP and tells how to discover advertisements using the Java reference implementation of JXTA.

Introducing the Peer Discovery Protocol

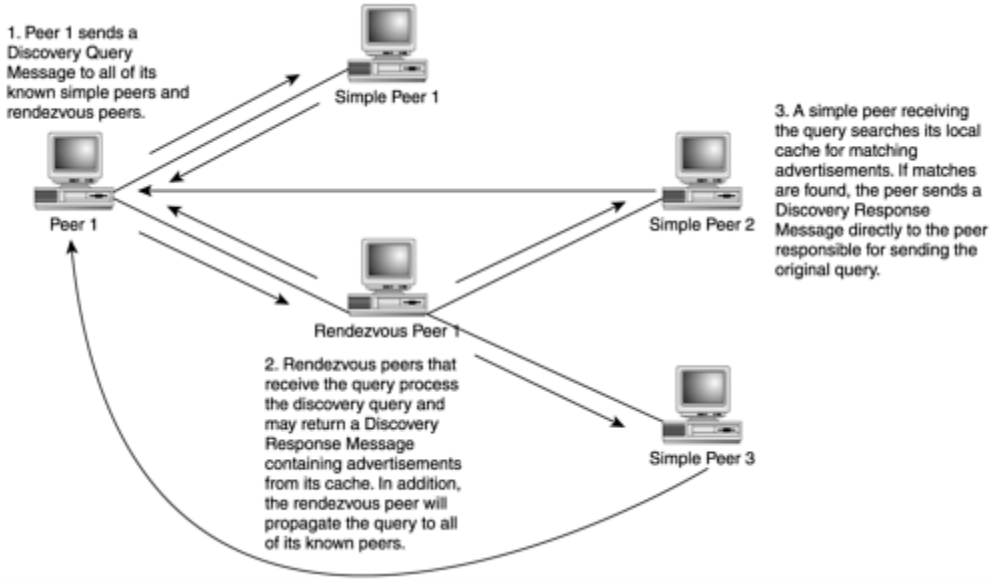
In Chapter 2, you saw that peers discover resources by sending a request to another peer, usually a rendezvous peer, and receiving responses containing advertisements describing the available resources on the P2P network.

The Peer Discovery Protocol consists of only two messages that define the following:

- A request format to use to discover advertisements
- A response format for responding to a discovery request

These two message formats, the Discovery Query Message and the Discovery Response Message, define all the elements required to perform a discovery transaction between two peers, as shown in Figure 4.1.

Figure 4.1. Exchange of discovery messages.



Although the messages define a request and a response to that request, it is important to note that a peer might not expect a Discovery Response Message in response to a given Discovery Query Message. A response to a request might not be received for a variety of reasons—for example, the request didn't generate any results, or the request was ignored by an overloaded peer.

The Discovery Query Message

The Discovery Query Message is sent to other peers to find advertisements. It has a simple format, as shown in Listing 4.1.

Listing 4.1 The Discovery Query Message XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:DiscoveryQuery>
  <Type> . . . </Type>
  <Threshold> . . . </Threshold>
  <PeerAdv> . . .</PeerAdv>
  <Attr> . . . </Attr>
  <Value> . . .</Value>
```

```
</jxta:DiscoveryQuery>
```

The root element for the Discovery Query Message is the `jxta:DiscoveryQuery` element. Developers familiar with XML might recognize the `jxta:` prefix in the root element as an XML namespace specifier and wonder if the `jxta` namespace is used or enforced within the Java implementation. Although the `jxta` prefix does specify a namespace, the current Java implementation of JXTA does not understand XML namespaces and treats `jxta:DiscoveryQuery` as the element name rather than recognizing `DiscoveryQuery` as an XML tag from the `jxta` namespace.

The elements of the Discovery Query Message describe the discovery parameters for the query. Only advertisements that match all the requirements described by the query's discovery parameters are returned by a peer. The discovery parameters described by the Discovery Query Message are listed here:

- **`Type`**— A required element containing an integer value specifying the type of advertisement being discovered. A value of `0` represents a query for Peer Advertisements, `1` represents a query for Peer Group Advertisements, and `2` represents a query for any other type of advertisement.

- **`Threshold`**— An optional element containing a number specifying the maximum number of advertisements that should be sent by a peer responding to the query.

- **`PeerAdv`**— An optional element containing the Peer Advertisement for the peer making the discovery query. The Peer Advertisement contains details that uniquely identify the peer on the network to enable another peers to respond to the query.

- **`Attr` and `Value`**— An optional pair of elements that together specify the criteria that an advertisement must fulfill to be returned as a response to this query. `Attr` specifies the name of an element, and `Value` specifies the value that the element must have to be returned as a response to the query.

A couple special exceptions to these rules apply:

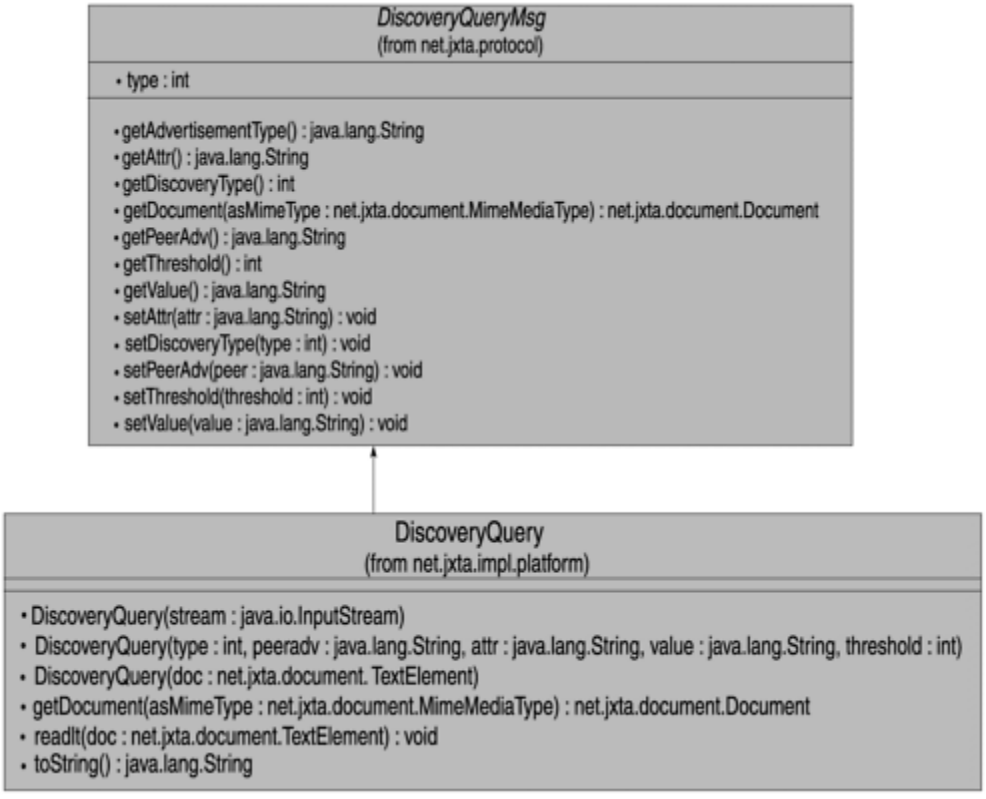
- When the `Type` is set to `0` (representing a query for Peer Advertisements) and the threshold is set to `0`, the peer sending the Discovery Query Message is seeking to

obtain as many Peer Advertisements as possible. All peers that receive the query should respond to the query with their Peer Advertisement.

- When values for the `Attr` and `Value` elements are absent, each peer responds with a random set of advertisements of the requested `Type`,up to the maximum specified by the `Threshold` element.

In the Java reference implementation, the Discovery Query Message's definition is split into an abstract class definition and a reference implementation provided by Project JXTA, as shown in Figure 4.2. The purpose of this division is to allow third-party developers to maintain API compatibility with the Java reference implementation when providing their own implementation for message parsing and formatting.

Figure 4.2. The Discovery Query Message classes.



The abstract definition of the Discovery Query Message can be found in the `net.jxta.protocol.DiscoveryQueryMsg` class, and the reference implementation of the abstract class can be found in the `net.jxta.impl.protocol.DiscoveryQuery` class.

Listing 4.2 provides the `shell` command to create a Discovery Query Message using the `DiscoveryQuery` implementation and prints it to the Shell's standard output for examination.

Listing 4.2 Source Code for *example4_1.java*

```
package net.jxta.impl.shell.bin.example4_1;

import java.io.StringWriter;

import net.jxta.document.StructuredTextDocument;
import net.jxta.document.MimeMediaType;
import net.jxta.discovery.DiscoveryService;
import net.jxta.impl.protocol.DiscoveryQuery;
import net.jxta.impl.shell.ShellApp;

/**
 * A Shell command to create and output a Discovery Query Message.
 */
public class example4_1 extends ShellApp
{

    /**
     * The implementation of the Shell command, invoked when the command
     * is started by the user from the Shell.
     *
     * @param   args the command-line arguments passed to the command.
     * @return  a status code indicating the success or failure of
     *     the command.
     */
    public int startApp(String[] args)
    {
        int result = appNoError;

        int type = DiscoveryService.PEER;
        String attribute = null;
```

```
        String value = null;
        int threshold = 0;
        String advertisementString = "This is my Peer Advertisement";

        // Construct a discovery query message.
        DiscoveryQuery query =
            new DiscoveryQuery(type, advertisementString, attribute,
                value, threshold);

        // Create an XML formatted string version of the discovery query.
        StringWriter buffer = new StringWriter();
        MimeMediaType mimeType = new MimeMediaType("text/xml");
        // MimeMediaType mimeType = new MimeMediaType("text/plain");
        try
        {
              StructuredTextDocument document =
                  (StructuredTextDocument) query.getDocument(mimeType);
              document.sendToWriter(buffer);
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

        // Print out the formatted message.
        println(buffer.toString());

        return result;
    }
}
```

Place the example's code in a file called `example4_1.java` in the `Shell` subdirectory of the JXTA demo installation. Compile the example using this code:

```
javac -d . -classpath ..\lib\beepcore.jar;..\lib\cms.jar;
..\lib\cryptix-asn1.jar;..\lib\cryptix32.jar;..\lib\instantp2p.jar;
```

```
..\lib\jxta.jar;..\lib\jxtaptls.jar;..\lib\jxtasecurity.jar;
..\lib\jxtashell.jar;..\lib\log4j.jar;..\lib\minimalBC.jar
example4_1.java
```

After the example has compiled, run the Shell application using this code:

```
java -classpath ..\lib\beepcore.jar;..\lib\cms.jar;
..\lib\cryptix-asn1.jar;..\lib\cryptix32.jar;..\lib\instantp2p.jar;
..\lib\jxta.jar;..\lib\jxtaptls.jar;..\lib\jxtasecurity.jar;
..\lib\jxtashell.jar;..\lib\log4j.jar;..\lib\minimalBC.jar;.
 net.jxta.impl.peergroup.Boot
```

When the Shell has loaded, run the example using this command:

```
JXTA>example4_1
```

The `example4_1` command produces the XML-formatted Discovery Query Message containing the parameters for the discovery, shown in Listing 4.3.

Listing 4.3 Output of the *example4_1* Shell Command

```
<?xml version="1.0"?>

<!DOCTYPE jxta:DiscoveryQuery>

<jxta:DiscoveryQuery xmlns:jxta="http://jxta.org">
      <Type>
            0
      </Type>
      <Threshold>
            0
      </Threshold>
      <PeerAdv>
            This is my Peer Advertisement
      </PeerAdv>
</jxta:DiscoveryQuery>
```

The `DiscoveryQuery` constructor uses a `String` representation for the Peer Advertisement instead of an object, and the `String` passed to the constructor is used directly in the output. The Discovery Query Message output produced by the example isn't valid because the `PeerAdv` element doesn't actually contain a valid Peer Advertisement. Producing a valid Discovery Query Message using the `DiscoveryQuery` class requires the developer to create a Peer Advertisement object and format it as a `String` in the same manner that the example uses to create a `String` from the `query` object. This `String` then set as the contents of `PeerAdv` element using `DiscoveryQuery` 's `setPeerAdv` method. For now, you'll avoid creating the Peer Advertisement object; we'll focus on it later in this chapter when advertisement instantiation is explored.

The mechanism for formatting the `query` object as a `String` is entirely abstracted through the `net.jxta.document.Document` interface. The `Document` interface defines a generic container for MIME media that can be read from an `InputStream` or written to an `OutputStream`. All advertisement and message objects used by the Java implementation of JXTA use an implementation of the `StructuredTextDocument` interface, derived from the `Document` interface, to provide a representation of the class as a structured MIME text document.

Using the `StructuredTextDocument` interface, the `query` object in the example is written out to XML by providing a `MimeMediaType` object for the `text/xml` MIME type to the `query` object's `getDocument` method. Because the formatting framework is so flexible, the output format could be easily changed to print plain text instead of XML by changing the following line in the example:

```
MimeMediaType mimeType = new MimeMediaType("text/xml");
```

To print plain text, create a `MimeMediaType` object for the `text/plain` MIME type instead of `text/xml` using the following line:

```
MimeMediaType mimeType = new MimeMediaType("text/plain");
```

When this change is in place, recompile the example and restart the Shell application. Running the `example4_1` command this time produces the result shown in Listing 4.4.

Listing 4.4 Output of the Modified *example4_1* Shell Command

```
jxta:DiscoveryQuery :
      Type : 0
      Threshold : 0
      PeerAdv : This is my Peer Advertisement
```

The format of the output is determined by the `MimeMediaType` object passed to `getDocument`. The `query` object's `getDocument` method uses this MIME type and the `StructuredDocumentFactory` to produce an implementation of the `StructuredDocument` interface. The available implementations of `StructuredDocument` are defined in the `StructuredDocumentInstanceTypes` property of the `config.properties` property file located in the `net.jxta.impl` package. Currently, only two implementations, `LiteXMLDocument` and `PlainTextDocument`,are available, corresponding to the `text/xml` and `text/plain` MIME types, respectively. The abstraction of message and advertisement formatting means that the Java reference implementation could switch easily from XML to another, possibly binary, format without requiring major changes to the implementation architecture.

The example demonstrates only how to create a Discovery Query Message, not how to send it to other peers to perform the actual discovery. An application developer never actually needs to formulate a Discovery Query Message and send it to other peers themselves; in fact, there is no abstract way of instantiating a `DiscoveryQueryMsg` implementation in the Java reference implementation. The `DiscoveryQueryMsg` is an abstract class defining an interface that `DiscoveryQuery` implements. Although a developer can use the `DiscoveryQuery` implementation directly, this prevents a developer from using another implementation without changing all the code. As you'll see, developers discover advertisements using the Discovery service instead of using the `DiscoveryQueryMsg` class or its implementations directly, thereby abstracting the developer from a particular implementation of `DiscoveryQueryMsg`.

The Discovery Response Message

To reply to a Discovery Query Message, a peer creates a Discovery Response Message that contains advertisements that match the query's search criteria, such as the `Attr` / `Value`

combination or `Type` of advertisement. The Discovery Response Message is formatted as shown in Listing 4.5.

Listing 4.5 The Discovery Response Message XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:DiscoveryResponse>
  <Type> . . . </Type>
  <Count> . . . </Count>
  <PeerAdv> . . . </PeerAdv>
  <Attr> . . . </Attr>
  <Value> . . . </Value>
  <Response Expiration="expiration time">
    . . .
  </Response>
</jxta:DiscoveryResponse>
```

The elements of the Discovery Response Message closely correspond to those of the Discovery Query Message:

- **`Type`**— Similar to the `Type` element passed in the Discovery Query Message, the `Type` element here is a required element containing an integer value that represents the type of all the advertisements contained within the `Response` elements of the message. As before, a value of `0` represents Peer Advertisements, `1` represents Peer Group Advertisements, and `2` represents all other types of advertisements.

- **`Count`**— An optional element containing an integer representing the total number of `Response` elements in the message.

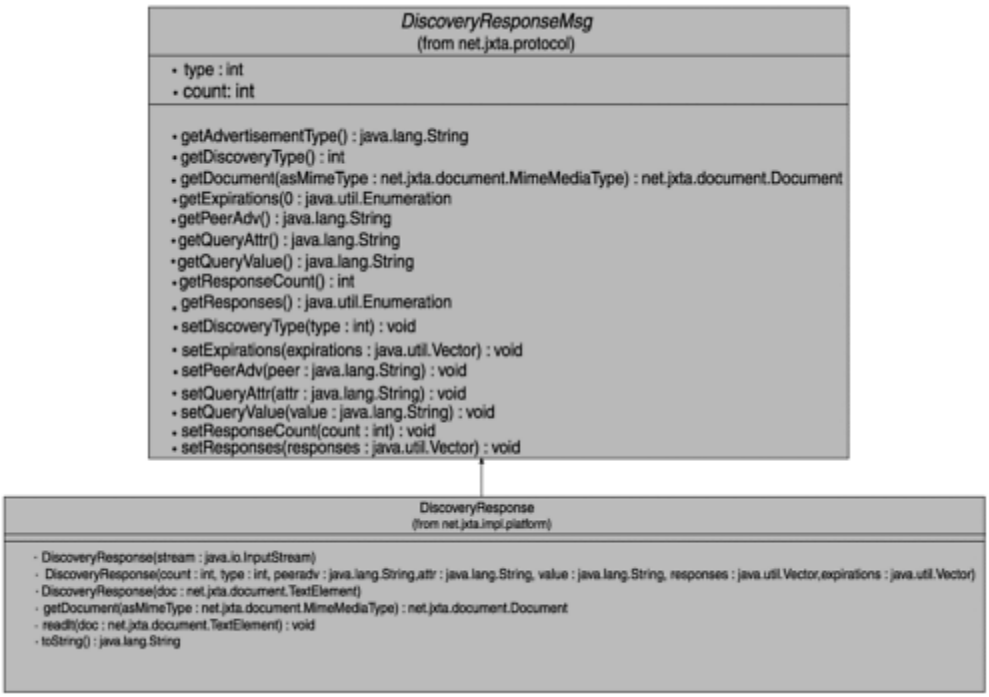
- **`PeerAdv`**— An optional element containing the Peer Advertisement of the peer responding to the original Discovery Query Message.

- **`Attr` and `Value`**— An optional pair of elements that together specify the original search criteria that generated this response. These have the same value as the `Attr` and `Value` in the Discovery Query Message; if these elements were not present in the original query, they are omitted from the response.

- **`Response`**— An optional element containing an advertisement that matched the search criteria in the Discovery Query Message. Each Discovery Response Message can contain multiple `Response` elements, each containing one advertisement in response to the original query. The total number of `Response` elements equals the value held by the `Count` element. The `Expiration` attribute on the `Response` elements specifies the length of time that this advertisement should be considered valid. In the Java reference implementation, this time is implicitly expressed in milliseconds.

The abstract definition of the Discovery Response Message is defined in the `net.jxta.protocol.DiscoveryResponseMsg` class, shown in Figure 4.3, and the reference implementation is defined in the `net.jxta.impl.protocol.DiscoveryResponse` class.

Figure 4.3. The Discovery Response Message classes.



Unlike with `DiscoveryQueryMsg` class, a developer uses the `DiscoveryResponseMsg` class in conjunction with the Discovery service to process responses to queries. The `DiscoveryResponseMsg` class provides developers with an easy mechanism to extract response advertisements; this is demonstrated in the example in the next section.

The Discovery Service

All the protocols defined by the JXTA Protocols Specification are implemented as services called *core services*. The core services include the following:

- Discovery
- Pipe
- Endpoint
- Rendezvous
- Peer Info
- Resolver

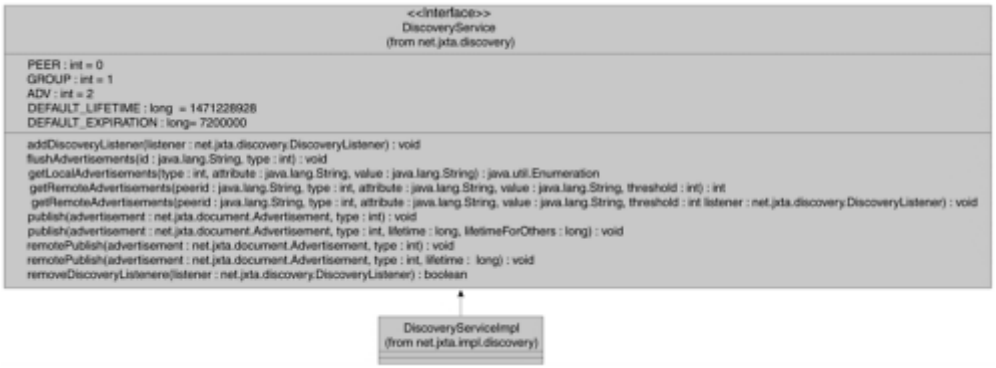
An instance of a service is associated with a specific peer group. Only peers that are members of the same peer group are capable of communicating with each other via their services. By default, all peers belong to a common peer group, called *Net Peer Group*, thereby allowing all peers and their advertisements to be discovered.

Services provide developers with a level of abstraction, insulating them somewhat from the raw message objects used to send information between peers. The Discovery service provides a mechanism for the following:

- Retrieving remote advertisements
- Retrieving local advertisements
- Publishing advertisements locally
- Publishing advertisements remotely
- Flushing local advertisements

In the Java reference implementation, the Discovery service is defined by the `DiscoveryService` interface in `net.jxta.discovery` and is implemented by the `DiscoveryServiceImpl` class in net. `jxta.impl.discovery`, as shown in Figure 4.4.

Figure 4.4. The `DiscoveryService` interface and implementation.



The `DiscoveryService` interface provides a simple mechanism for developers to send discovery queries and process discovery responses. A small set of convenience methods allows developers to send Discovery Query Messages without requiring the developer to create and populate a `DiscoveryQuery` object beforehand.

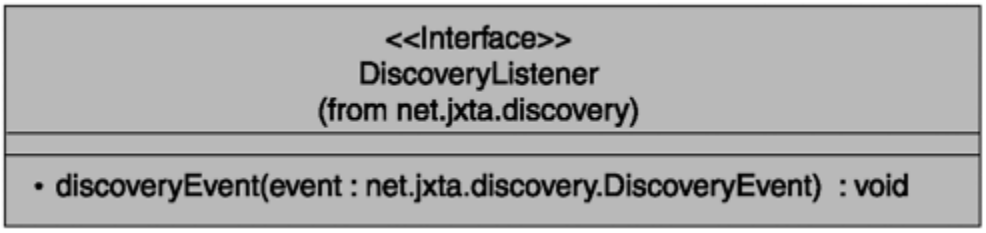
The DiscoveryListener Interface

An application requires some way of being notified of responses to a discovery query to allow the application to extract advertisements from the response. In the Java reference implementation, developers can register a *listener* object that will be notified by the `DiscoveryService` when Discovery Response Messages are received.

Java developers are probably most familiar with the concept of a listener from the Java Foundation Classes (JFC). In the JFC, a listener interface is defined for each type of event that can be generated from a user interface widget, such as a button. An object that wants to be informed when a button is clicked implements the appropriate listener interface and registers itself with the button. When the button is clicked, the button widget calls the appropriate method of each listener implementation instance that has registered with the widget.

The Java reference implementation uses a similar mechanism to allow developers to be informed when a new Discovery Response Message is received by the `DiscoveryService`. A developer wanting to be notified of the arrival of a new Discovery Response Message needs to create an implementation of the `DiscoveryListener` interface, as shown in Figure 4.5.

Figure 4.5. The `DiscoveryListener` interface.



To receive notification, the developer registers the implementation of the `DiscoveryListener` interface with an instance of the `DiscoveryService` using the `addDiscoveryListener` method defined in the `net.jxta.discovery.Discovery` interface:

```
public void addDiscoveryListener(
    DiscoveryListener listener);
```

Each time the `DiscoveryService` instance receives a Discovery Response Message, the listener's `discoveryEvent` method is called with an event detailing the response received by the service.

To stop receiving notifications, the listener object must be removed from the `DiscoveryService` using the `removeDiscoveryListener` method defined in the `net.jxta.discovery.Discovery` interface:

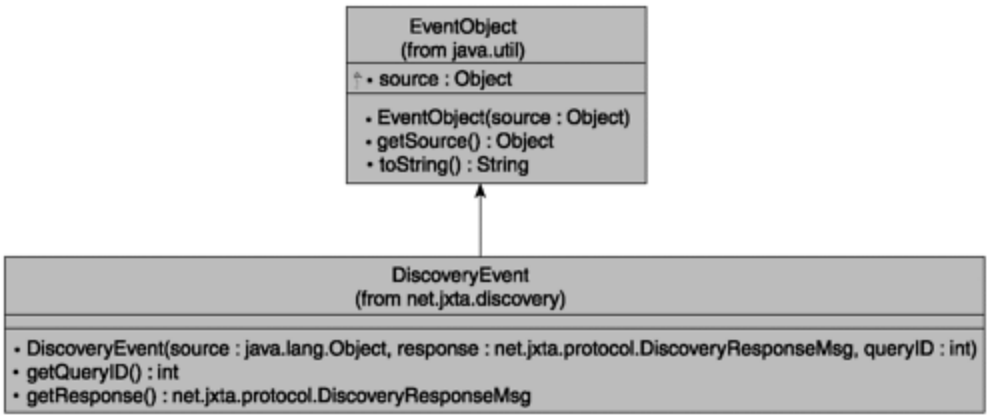
```
public boolean removeDiscoveryListener(DiscoveryListener listener);
```

A reference to the original listener object is required to be capable of removing the listener object from the `DiscoveryService` instance. The call to the `removeDiscoveryListener` returns `true` if the given `listener` object is removed from the `DiscoveryService` instance, or `false` if the `listener` object isn't currently registered with the `DiscoveryService` instance.

The DiscoveryEvent Class

As shown in Figure 4.6, the `DiscoveryEvent` defined in `net.jxta.discovery` is provided to the `discoveryEvent` method of the `DiscoveryListener` implementation to provide details on the Discovery Response Message received by a `DiscoveryService` instance.

Figure 4.6. The `DiscoveryEvent` class.



The listener can extract the `DiscoveryResponseMsg` from the event using the `getResponse` method of `DiscoveryEvent`:

```
public DiscoveryResponseMsg getResponse()
```

Use the `getResponses` method of `DiscoveryResponseMsg`, as shown in Listing 4.6, to obtain an `Enumeration` object that can be used to iterate over the advertisements returned in the `DiscoveryResponseMsg`.

Listing 4.6 Extracting Responses from a *DiscoveryEvent* Object

```
public void discoveryEvent(DiscoveryEvent event)
{
    DiscoveryResponseMsg response = event.getResponse();
    Enumeration enum = response.getResponses();
```

```
    while (enum.hasMoreElements())
    {
        String advString =
            (String) enum.nextElement();

        // Extract the advertisement from the string here.
    }
}
```

The `DiscoveryResponseMsg` interface also provides the `getExpirations` method, allowing a developer to obtain an `Enumeration` of the expiration times for each of the advertisements returned in the response.

Using DiscoveryListener and DiscoveryEvent

To try out handling discovery responses, you'll create a `shell` command to handle registering and unregistering your own `DiscoveryListener` implementation. First, you need an implementation of the `DiscoveryListener` interface, as shown in Listing 4.7.

Listing 4.7 Source Code for *ExampleListener.java*

```
package net.jxta.impl.shell.bin.example4_2;

import java.util.Enumeration;

import net.jxta.document.Advertisement;
import net.jxta.discovery.DiscoveryEvent;
import net.jxta.discovery.DiscoveryListener;
import net.jxta.protocol.DiscoveryResponseMsg;


/**
 * A simple listener to notify the user when a discovery event has
 * been received.
 */
public class ExampleListener implements DiscoveryListener
```

```
{
    /**
     * The DiscoveryListener's event method, used for handling
     * notification of a received Discovery Response Message from
     * the Discovery service.
     *
     * @param   event the event containing the received response.
     */
    public void discoveryEvent(DiscoveryEvent event)
    {
        DiscoveryResponseMsg response = event.getResponse();

        System.out.println("Received a response containing "
            + response.getResponseCount() + " advertisements");
    }
```

For this simple example, you don't need anything fancy—just a notification that a response has been received and details on the number of advertisements contained in the response. Next, you need to create a Shell command called `example4_2` to handle registering and unregistering your listener object. This is shown in Listing 4.8.

Listing 4.8 Source Code for *example4_2.java*

```
package net.jxta.impl.shell.bin.example4_2;

import net.jxta.discovery.DiscoveryService;
import net.jxta.discovery.DiscoveryListener;

import net.jxta.peergroup.PeerGroup;

import net.jxta.impl.shell.GetOpt;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;
```

```
package net.jxta.impl.shell.bin.example4_2;
import net.jxta.discovery.DiscoveryService;
import net.jxta.discovery.DiscoveryListener;

import net.jxta.peergroup.PeerGroup;

import net.jxta.impl.shell.GetOpt;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;


/**
 * A simple example shell application to register or unregister a
 * DiscoveryListener.
 */
public class example4_2 extends ShellApp
{
    /**
     * The shell environment holding the store of environment variables.
     */
    ShellEnv theEnvironment;

    /**
     * A flag indicating whether to add or remove the listener.
     */
    boolean addListener = true;

    /**
     * The name used to store the listener in the environment.
     */
    String name = "Default";


    /**
     * Manages adding or removing the listener.
```

```
 *
 * @param   discovery the Discovery service to use to manage
 *          the listener.
 */
private void manageListener(DiscoveryService discovery)
{
    if (name != null)
    {
        // Check if a listener already exists.
        ShellObject theShellObject = theEnvironment.get(name);

        if (addListener)
        {
            if (theShellObject == null)
            {
                // Create a new listener.
                DiscoveryListener listener = new ExampleListener();

                // Add the listener to the discovery service.
                discovery.addDiscoveryListener(listener);

                // Add the listener object to the environment.
                theEnvironment.add(name,
                    new ShellObject(name, listener));
            }
        }
        else
        {
            if (theShellObject != null)
            {
                DiscoveryListener listener =
                    (DiscoveryListener) theShellObject.getObject();
                // Remove the listener from the discovery service.
                discovery.removeDiscoveryListener(listener);

                // Remove the listener object from the environment.
```

```
                theEnvironment.remove(name);
            }
        }
    }
}
/**
 * Parses the command-line arguments and initializes the command
 *
 * @param       args the arguments to be parsed.
 * @exception   IllegalArgumentException if an invalid parameter
 *              is passed.
 */
private void parseArguments(String[] args)
    throws IllegalArgumentException
{
    int option;

    // Parse the arguments to the command.
    GetOpt parser = new GetOpt(args, "rn:");

    while ((option = parser.getNextOption()) != -1)
    {
        switch (option)
        {
            case 'r' :
            {
                // Remove the listener.
                addListener = false;
                break;
            }

            case 'n' :
            {
                // Get the name used to store the listener object.
                String argument= null;
```

```
                if ((argument = parser.getOptionArg()) != null)
                {
                    name = argument;
                }

                break;
            }
        }
    }
}

/**
 * The implementation of the Shell command, invoked when the command
 * is started by the user from the Shell.
 *
 * @param   args the command-line arguments passed to the command.
 * @return  a status code indicating the success or failure of
 *          the command.
 */
public int startApp(String[] args)
{
    int result = appNoError;

    // Get the shell's environment.
    theEnvironment = getEnv();

    // Use the environment to obtain the current peer group.
    ShellObject theShellObject = theEnvironment.get("stdgroup");
    PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

    // Get the Discovery service for the current peer group.
    DiscoveryService discovery = currentGroup.getDiscoveryService();

    try
    {
        // Parse the command-line arguments.
```

```
            parseArguments(args);
        }
        catch (IllegalArgumentException e)
        {
            println("Incorrect parameters passed to the command.");
            result = ShellApp.appParamError;
        }

        // Manage the listener to the Discovery service. This
        // adds or removes the listener as specified by the
        // command-line parameters.
        manageListener(discovery);

        return result;

    }
}
```

By default, the `example4_2` command creates a listener, adds it to the current peer group's `DiscoveryService`, and stores the listener in a Shell environment variable named `Default`. Storing the listener object is essential; otherwise, the listener can't be removed from the `DiscoveryService` at a later time.

Note

Even if you already configured the Shell in the past, you will be prompted each time you start the Shell to provide your username and password. When trying out the examples, this can become annoying. To avoid having to enter your username and password each time, you can pass in your username and password as system properties to the Java runtime. Use this command to pass in your `username` and `password` as system properties:

```
java -Dnet.jxta.tls.password=password
-Dnet.jxta.tls.principal=username . . .
```

This sets a system property called net.jxta.tls.password to the password value provided after the equals (=) sign and a system property called `net.jxta.tls.principal` to the

username provided. When you start the Shell from the command line and include these parameters, the Shell starts immediately without prompting for your username and password.

Place the source code in the `Shell` subdirectory of the JXTA installation and compile it in the same way that you compiled the previous example. Start the Shell from the command line. After the Shell has loaded, clear the local cache of Peer Advertisements using this line:

```
JXTA>peers -f
```

Register an `ExampleListener` instance by running the `example4_2` command:

```
JXTA>example4_2
```

You can check that a `Shell` variable has been created using the variable name `Default` by checking the output of the `env` command. At this point, a `DiscoveryListener` has been registered to be notified when Discovery Response Messages are received by the current peer group's `DiscoveryService`. The code responsible for retrieving the current peer group and the peer group's `DiscoveryService` is shown in Listing 4.9.

Listing 4.9 Obtaining DiscoveryService

```
// Get the shell's environment.
theEnvironment = getEnv();

// Use the environment to obtain the current peer group.
ShellObject theShellObject = theEnvironment.get("stdgroup");
PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

// Get the Discovery service for the current peer group.
DiscoveryService discovery = currentGroup.getDiscoveryService();
```

This code retrieves the current peer group's `PeerGroup` object from the Shell's environment, where it is always stored using the name `stdgroup`. This object obtains a reference to the `DiscoveryService` object that is used by the `manageListener` method to either add or remove the listener.

To see the listener in action, send a discovery query using the `peers -r` command:

```
JXTA>peers -r
```

Every time your peer receives responses to the query, the `ExampleListener` object's `discoveryEvent` method prints the number of advertisements in the response message:

```
Received a response containing 4 advertisements
```

This output appears not in the Shell itself, but in the standard output of the command shell used to start the Shell application. Although you could print the output to the Shell console, it would require delving into the use of pipes, which isn't appropriate at this point.

Instead of sending an active discovery query, try using the `peers` command to retrieve local Peer Advertisements, and observe the behavior of the `ExampleListener` output.You should observe that the `ExampleListener` never receives notification of responses to a local discovery query. The `DiscoveryService` uses the local cache to provide immediate responses to a call to send a local discovery query; therefore, registered listeners never receive a notification of a response to a local discovery query.

The `example4_2` command takes two optional parameters, `-r` and `-n`. The `-r` option indicates to the command that the listener object should be removed from the `DiscoveryService`, and the `-n` option indicates the name of the variable storing the listener instance. For example, issuing the following line attempts to retrieve a `DiscoveryListener` object from an environment variable named `MyListener` and remove the retrieved listener object from the `DiscoveryService` instance:

```
JXTA>example4_2 -r -nMyListener
```

The arguments to the `example4_2` command are parsed easily using the `GetOpt` object in the example:

```
GetOpt parser = new GetOpt(args, "rn:");
```

The second argument to the `GetOpt` constructor, called the *format string*, specifies the command's options and whether the option has any arguments. If a character is followed by the `:` (colon) character, that option requires an argument; if it is followed by the `;` (semicolon), the option has an optional argument. This functionality will be used again in later examples.

At this point, you know how to receive notification of a response to discovery query but not how to send the actual discovery query itself. The next section provides an example of how to send a discovery query to a remote peer using the `DiscoveryService`.

Finding Remote Advertisements

Rather than force developers to create a `DiscoveryQueryMsg` instance themselves, the `DiscoveryService` interface provides an easy way for developers to send a Discovery Query Message to other peers using the `getRemoteAdvertisements` method:

```
public int getRemoteAdvertisements (String peerid,
    int type, String attribute, String value,
    int threshold, DiscoveryListener listener);
```

Each parameter passed to `getRemoteAdvertisements` corresponds to a field in the Discovery Query Message, with the exception of the `peerid` and `listener` parameters. The `peerid` parameter is a parameter that uniquely identifies the peer to query for advertisements; if this parameter is `null`, the message is sent to all peers on the local network and is propagated via available rendezvous peers. More information on identifiers is provided in the section "Working with Advertisements" later in this chapter.

The `listener` parameter provides a `DiscoveryListener` object that is called only when responses arrive in response to this particular call to `getRemoteAdvertisements`.

Providing a `listener` object provides a way to receive notification without registering a listener with the `DiscoveryService`. Registered listeners are notified of incoming responses regardless of whether a `null` or non`null` listener is passed to `getRemoteAdvertisements`.

To try out the `getRemoteAdvertisements` method, the following example `shell` command shown in Listing 4.10 allows a user to send remote queries and specify the desired advertisement type and maximum responses.

Listing 4.10 Source Code for *example4_3.java*

```
package net.jxta.impl.shell.bin.example4_3;

import java.io.IOException;

import net.jxta.discovery.DiscoveryService;
import net.jxta.peergroup.PeerGroup;

import net.jxta.impl.shell.GetOpt;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;

/**
 * A simple example shell application to enable a user to send remote
 * discovery queries using the current peer group's Discovery service.
 */
public class example4_3 extends ShellApp
{
    /**
     * The type of advertisement to discover. Defaults to peer
     * advertisements.
     */
    private int type = DiscoveryService.PEER;

    /**
```

```
 * The maximum number of responses requested.
 */
private int threshold = 10;

/**
 * The Discovery service being used to discover advertisements.
 */
private DiscoveryService discovery = null;

/**
 * Parses the command-line arguments and initializes the command
 *
 * @param       args the arguments to be parsed.
 * @exception   IllegalArgumentException if an invalid parameter
 *               is passed.
 */
private void parseArguments(String[] args)
    throws IllegalArgumentException
{
    int option;
    // Parse the arguments to the command.
    GetOpt parser = new GetOpt(args, "a:t:");

    while ((option = parser.getNextOption()) != -1)
    {
        switch (option)
        {
            case 'a' :
            {
                // Set the type of advertisement to discover.
                type = Integer.parseInt(parser.getOptionArg());

                // Validate the type.
                if ((type < 0) || (type > 2))
                {
                    // Default to the peer type.
```

```
                    type = DiscoveryService.PEER;
                }

                break;
            }

            case 't' :
            {
                String argument = null;

                if ((argument = parser.getOptionArg()) != null)
                {
                    // Set the threshold.
                    threshold = Integer.parseInt(argument);
                }

                break;
            }
        }
    }
}

/**
 * Send a discovery request to remote peers via the Discovery service.
 *
 * @param   type the type of advertisement to discover.
 * @param   threshold the maximum number of advertisements to be
 *          returned by any single peer.
 */
private void sendRemoteDiscovery(int type, int threshold)
{
    discovery.getRemoteAdvertisements(null, type, null, null,
        threshold, null);
}

/**
```

```
 * The implementation of the Shell command, invoked when the command
 * is started by the user from the Shell.
 *
 * @param   args the command-line arguments passed to the command.
 * @return  a status code indicating the success or failure of
 *          the command.
 */
public int startApp(String[] args)
{
    int result = appNoError;

    // Get the shell's environment.
    ShellEnv theEnvironment = getEnv();

    // Use the environment to obtain the current peer group.
    ShellObject theShellObject = theEnvironment.get("stdgroup");
    PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

    // Get the Discovery service for the current peer group.
    discovery = currentGroup.getDiscoveryService();

    try
    {
        // Parse the command-line arguments.
        parseArguments(args);
    }
    catch (IllegalArgumentException e)
    {
        println("Incorrect parameters passed to the command.");
        result = ShellApp.appParamError;
    }
    // Send a remote discovery request.
    sendRemoteDiscovery(type, threshold);

    return result;
}
```

```
}
```

The example is essentially a replacement for the `peers -r` command. When run in conjunction with `example4_2`, it allows a user to send queries and be notified when responses arrive.

To see the `example4_3` command in action, first register a listener using the `example4_2` command:

```
JXTA>example4_2
```

Then send a Discovery Query Message that searches for Peer Advertisements, with a maximum of 10 responses from any given peer:

```
JXTA>example4_3
```

The peer sends a Discovery Query Message to all known peers requesting a response containing matching advertisements. The `ExampleListener` registered using the `example4_2` command prints information each time that a response to this query is received by the `DiscoveryService` instance.

Finding Cached Advertisements

In the Java reference implementation, advertisements in responses to a Discovery Query Message are automatically added to a local cache of advertisements. `DiscoveryListener` implementations don't have to provide caching functionality themselves.

To find advertisements using the local cache, a developer can use the `getLocalAdvertisements` method of the `DiscoveryService` interface. Unlike performing an active discovery to find advertisements on remote peers, performing discovery using the local cache returns results immediately and does not require an implementation of the `DiscoveryListener` interface.

To see how local discovery works, Listing 4.11 shows another example Shell command that replaces some of the functionality of the `peers` command.

Listing 4.11 Source Code for *example4_4.java*

```
package net.jxta.impl.shell.bin.example4_4;

import java.io.IOException;

import java.util.Enumeration;

import net.jxta.discovery.DiscoveryService;

import net.jxta.document.Advertisement;

import net.jxta.peergroup.PeerGroup;

import net.jxta.impl.shell.GetOpt;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;



/**
 * A simple example shell application to enable a user to send local
 * discovery queries using the current peer group's Discovery service.
 */
public class example4_4 extends ShellApp
{
    /**
     * The type of advertisement to discover. Defaults to
     * peer advertisements.
     */
    private int type = DiscoveryService.PEER;

    /**
     * The Discovery service being used to discover advertisements.
     */
    private DiscoveryService discovery = null;
```

```
/**
 * The name of the element to match.
 */
private String attribute = null;
/**
 * The value to match for the element specified by the attribute
 * variable.
 */
private String value = null;



/**
 * Parses the command-line arguments and initializes the command
 *
 * @param       args the arguments to be parsed.
 * @exception   IllegalArgumentException if an invalid parameter
 *      is passed.
 */
private void parseArguments(String[] args)
    throws IllegalArgumentException
{
    int option;

    // Parse the arguments to the command.
    GetOpt parser = new GetOpt(args, "a:k:v:");

    while ((option = parser.getNextOption()) != -1)
    {
        switch (option)
        {
            case 'a' :
            {
                // Set the type of advertisement to discover.
                type = Integer.parseInt(parser.getOptionArg());
```

```
                // Validate the type.
                if ((type < 0) || (type > 2))
                {
                    // Default to the peer type.
                    type = DiscoveryService.PEER;
                }

                break;
            }
            case 'k' :
            {
                // Set the attribute to match.
                attribute = parser.getOptionArg();

                break;
            }

            case 'v' :
            {
                // Set the value for the attribute being matched.
                value = parser.getOptionArg();

                break;
            }
        }
    }

    // Both attribute and value must be specified.
    if (!((null != attribute) && (null != value)))
    {
        // Set both to null.
        attribute = null;
        value = null;
    }
}
```

```
    /**
     * Sends a local discovery request using the Discovery service.
     */
    private void sendLocalDiscovery()
    {
        try
        {
            int count = 0;
            Enumeration enum =
                discovery.getLocalAdvertisements(type, attribute,
value);
            Advertisement advertisement;

            // Iterate through the response advertisements.
            while (enum.hasMoreElements())
            {
                // Get the next element from the enumeration.
                advertisement = (Advertisement) enum.nextElement();

                println("Found a matching advertisement!");

                // Increment the counter.
                count++;
            }

            println("Found " + count + " advertisements!");
        }
        catch (IOException e)
        {
            println("Error discovering local advertisements!" + e);
        }
    }

    /**
     * The implementation of the Shell command, invoked when the command
     * is started by the user from the Shell.
```

```
     *
     * @param   args the command-line arguments passed to the command.
     * @return  a status code indicating the success or failure of
     *          the command.
     */
    public int startApp(String[] args)
    {
        int result = appNoError;

        // Get the shell's environment.
        ShellEnv theEnvironment = getEnv();

        // Use the environment to obtain the current peer group.
        ShellObject theShellObject = theEnvironment.get("stdgroup");
        PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

        // Get the Discovery service for the current peer group.
        discovery = currentGroup.getDiscoveryService();

        try
        {
        // Parse the command-line arguments.
        parseArguments(args);
    }
    catch (IllegalArgumentException e)
    {
        println("Incorrect parameters passed to the command.");
        result = ShellApp.appParamError;
    }

    // Send a local discovery request.
    sendLocalDiscovery();

    return result;
}
```

Instead of using a `DiscoveryListener` implementation to handle advertisements returned in a `DiscoveryResponseMsg` response, `getLocalAdvertisements` returns an `Enumeration` of advertisements immediately that match the query parameters provided.

This example allows the user to provide an attribute and value to match, allowing the user to search for an advertisement that matches specific criteria. For example, assume that the `peers` command returns the following:

```
peer0: name = Cadillac
peer1: name = spec
peer2: name = spiro
peer3: name = zynevich
```

The `example4_4` command could be used to search the local cache for any Peer Advertisement in which the peer has a given name. The name of a peer is described by the `Name` element in its advertisement and is displayed as the `name` in the list returned by the `peers` command. To discover Peer Advertisements in which the `Name` is spec using the `example4_4` command, do the following:

```
JXTA>example4_4 –a0 –kName -vspec
Found a matching advertisement!
Found 1 advertisements!
```

The `–a` option specifies the advertisement type to discover, and the `–k` and `–v` options together specify a tag and value that an advertisement must contain to be part of a peer's response to the query. Discovering an advertisement that matches a given tag and value combination can even use a wildcard in the value string. Discovering Peer Advertisements with a tag called `Name` whose value starts with the letter *s* could be accomplished as follows:

```
JXTA>example4_4 –a0 –kName –vs*
Found a matching advertisement!
Found 2 advertisements!
```

The wildcard symbol, `*`, can be used anywhere within the value term; however, the wildcard symbol can't be used by itself, and the value to be matched must consist of at least one nonwildcard character. Wildcards can even be used in multiple places in the search string:

```
JXTA>example4_4 -a0 -kName -v*ill*
Found a matching advertisement!
Found 1 advertisements!
```

The Cache Manager

The local cache, implemented by the Cache Manager class `Cm` in the `net.jxta.impl.cm` package, handles storing discovered advertisements in a local file and directory structure. The Cache Manager is responsible not only for providing search capabilities for local discovery requests, but also for finding advertisements that match Discovery Query Messages sent by other peers. The Cache Manager stores cached advertisements in a directory called `cm` under the current directory when the JXTA application is executed.

Flushing Advertisements

At some point, an application might need to clear the entire cache; this might be required when an application has not been used in a long time and all advertisements are suspected to be stale. As shown in Listing 4.12, the `DiscoveryService` provides a simple mechanism to allow an application to clear the cache of specific advertisement types that match a given type of advertisement and identifier string.

Listing 4.12 Source Code for *example4_5.java*

```
package net.jxta.impl.shell.bin.example4_5;

import java.io.IOException;

import net.jxta.discovery.DiscoveryService;

import net.jxta.peergroup.PeerGroup;
```

```
import net.jxta.impl.shell.GetOpt;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;


/**
 * A simple example shell application to enable a user to flush
 * discovered advertisements from the local cache using the current peer
 * group's Discovery service.
 */
public class example4_5 extends ShellApp
{
    /**
     * The type of advertisement to flush. Defaults to peer advertisements.
     */
    private int type = DiscoveryService.PEER;

    /**
     * The ID of the advertisement to flush. Defaults to null.
     */
    private String id = null;


    /**
     * Parses the command-line arguments and initializes the command
     *
     * @param       args the arguments to be parsed.
     * @exception   IllegalArgumentException if an invalid parameter
     *              is passed.
     */
    Private void parseArguments(String[] args)
        throws IllegalArgumentException
    {
        int option;
```

```
    // Parse the arguments to the command.
    GetOpt parser = new GetOpt(args, "a:i:");

    while ((option = parser.getNextOption()) != -1)
    {
        switch (option)
        {
            case 'a' :
            {
                // Set the type of advertisement to flush.
                type = Integer.parseInt(parser.getOptionArg());

                // Validate the type.
                if ((type < 0) || (type > 2))
                {
                    // Default to the peer type.
                    type = DiscoveryService.PEER;
                }

                break;
            }

            case 'i' :
            {
                // Set the ID string of the advertisement to flush.
                id = parser.getOptionArg();

                break;
            }
        }
    }
}

/**
 * The implementation of the Shell command, invoked when the command
```

```
     * is started by the user from the Shell.
     *
     * @param   args the command-line arguments passed to the command.
     * @return  a status code indicating the success or failure of
     *          the command.
     */
    public int startApp(String[] args)
    {
        int result = appNoError;

        // Get the shell's environment.
        ShellEnv theEnvironment = getEnv();

        // Use the environment to obtain the current peer group.
        ShellObject theShellObject = theEnvironment.get("stdgroup");
        PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

        // Get the DiscoveryService service for the current peer group.
        DiscoveryService discovery = currentGroup.getDiscoveryService();

        try
        {
            // Parse the command-line arguments.
            parseArguments(args);

            // Flush all of the advertisements of the given type and ID.
            discovery.flushAdvertisements(id, type);
        }
        catch (IllegalArgumentException e)
        {
            println("Incorrect parameters passed to the command.");
            result = ShellApp.appParamError;
        }
        catch (IOException e)
        {
            println("Error flushing advertisements: " + e);
```

```
            result = appMiscError;
        }

        return result;
    }
}
```

To remove all the Peer Advertisements from the local cache using the `example4_5` command, type the following:

```
JXTA>example4_5 -a0
```

Invoking the `peers` command after this command should return an empty list. To remove only a specific advertisement, the unique identifier for the advertisement is required; in the case of a Peer Advertisement, this identifier is given by the `PID` element. The `PID` can be obtained using the `cat` command to view a Peer Advertisement stored in the Shell's environment variables, as demonstrated in Chapter 3, "Introducing JXTA P2P Solutions." For example, imagine that the Peer Advertisement to be flushed from the cache has this ID:

```
urn:jxta:uuid-59616261646162614A78746150
32503323EC5B06B634476AB7418CB18BA45DCA03
```

It can be removed from the cache using this command:

```
JXTA> example4_5 -a0 -iurn:jxta:uuid-59616261646162614A7874615032503
323EC5B06B634476AB7418CB18BA45DCA03
```

Executing this command removes the advertisement from the cache by deleting its corresponding advertisement from within the Cache Manager's `cm` directory.

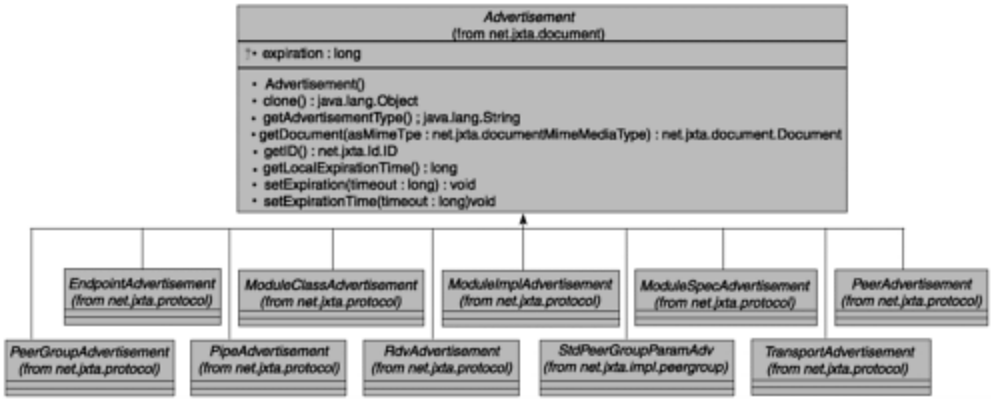
Working with Advertisements

At this point, this chapter has only really discussed advertisements in generic terms.You have not delved into the specifics of what information is contained by a particular

advertisement or how advertisements are used within the Java reference implementation. Although you know how to discover advertisements, how do you use them?

All advertisements in the Java reference implementation extend the `net.jxta.document.Advertisement` abstract class. The `Advertisement` class defines several methods, the most important being the `getDocument` method for transforming an `Advertisement` into a `Document` instance corresponding to a particular MIME type. As shown in Figure 4.7, each type of advertisement is split into an abstract class in the `net.jxta.protocol` package and an implementation class in `net.jxta.impl.protocol`.

Figure 4.7. The `Advertisement` abstract implementation classes.



The `net.jxta.protocol` abstract classes augment the `Advertisement` class with attributes specific to the type of advertisement and accessor methods to set and retrieve the value of those fields. The `net.jxta.impl.protocol` implementation classes provide the implementation of the `getDocument` method.

Instantiating an Advertisement

To insulate a developer from knowing about a specific advertisement implementation class, advertisements are instantiated using the `AdvertisementFactory` class in the `net.jxta.document` package. The simplest way to create an advertisement instance is to use the factory's static `newAdvertisement` method, providing a `String` containing the type of advertisement to create.

An advertisement type in the Java reference implementation is a `String` containing the root element of the advertisement that it is associated with. Although developers could construct the advertisement type `String` themselves, it is easier to use the static `getAdvertisementType` defined by the `Advertisement` class. For example, a `PeerAdvertisement` could be instantiated using either

```
PeerAdvertisement advertisement =
    (PeerAdvertisement)
        AdvertisementFactory.newAdvertisement(
            "jxta:PA");
```

or

```
PeerAdvertisement advertisement =
    (PeerAdvertisement)
        AdvertisementFactory.newAdvertisement(
            PeerAdvertisement.getAdvertisementType());
```

Each of the `net.jxta.protocol` subclasses of `Advertisement` provides an implementation of `getAdvertisementType` that allows a developer to get a specific type of advertisement without knowing which concrete class is providing the implementation.

Publishing Advertisements

Constructing an advertisement by itself doesn't make the advertisement known to either the local peer or any other peer on the network. For an advertisement to be available on the P2P network, it needs to be published locally, remotely, or both.

Publishing an advertisement locally places the advertisement in the local peer's cache of advertisements; other peers can find this advertisement using a standard Discovery Query Message. The `DiscoveryService` interface provides a simple mechanism for publishing the advertisement to the local cache using either

```
public void publish(Advertisement advertisement,
    int type) throws IOException;
```

or

```
public void publish (Advertisement adv, int type,
    long lifetime, long lifetimeForOthers)
        throws IOException;
```

The second version of the `publish` method is more explicit, allowing the caller to specify not only the advertisement and its type, but also the length of time that the advertisement will remain in the local cache and the length of time that the advertisement will be available to be discovered by other peers. The length of time in both cases is expressed in milliseconds, and the type of advertisement corresponds to the values used by the Discovery Query and Response Messages (`0` = peer, `1` = peer group, `2` = other advertisements).

The first version of the `publish` method publishes an advertisement to the local cache using default values for the local and remote lifetimes of the advertisement. The default local lifetime is one year, and the default lifetime for other peers is two hours.

To help accelerate the process of distributing an advertisement within the membership of a peer group, an advertisement can be published remotely. Publishing an advertisement remotely broadcasts the advertisement directly to other known peers or indirectly via known rendezvous peers to other members of the peer group associated with the `DiscoveryService` service instance. This broadcast uses a Discovery Response Message to push the advertisement to peers.

The `DiscoveryService` interface provides two methods, similar to the `publish` methods, to publish an advertisement to a remote peer. An advertisement can be remotely published using either

```
public void remotePublish (
    Advertisement adv, int type);
```

or

```
public void remotePublish (Advertisement adv, int
    type, long lifetime);
```

Although the documentation in the `DiscoveryService` interface specifies that the type can be set to indicate a peer, peer group, or other type of advertisement, the current implementation does not remotely publish Peer Advertisements. However, the reference implementation of `DiscoveryService`, `DiscoveryServiceImpl`, automatically adds the Peer Advertisement contained in any Discovery Query Messages that it receives, providing the same functionality.

To demonstrate the use of the `publish` and `remotePublish` methods, the `shell` command in Listing 4.13 creates a Peer Group Advertisement using the current peer group as a template, and publishes the advertisement locally and remotely.

Listing 4.13 Source Code for *example4_6.java*

```
package net.jxta.impl.shell.bin.example4_6;

import java.io.IOException;

import net.jxta.discovery.DiscoveryService;
import net.jxta.document.AdvertisementFactory;
import net.jxta.id.IDFactory;
import net.jxta.peergroup.PeerGroup;
import net.jxta.protocol.PeerGroupAdvertisement;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;


/**
 * A simple example shell application to publish a peer group
 * advertisement based on the Shell's current peer group.
 */
public class example4_6 extends ShellApp
{
    /**
     * The implementation of the Shell command, invoked when the command
     * is started by the user from the Shell.
```

```
     *
     * @param   args the command-line arguments passed to the command.
     * @return  a status code indicating the success or failure of
     *           the command.
     */
    public int startApp(String[] args)
    {
        int result = appNoError;

        // Get the shell's environment.
        ShellEnv theEnvironment = getEnv();

        // Use the environment to obtain the current peer group.
        ShellObject theShellObject = theEnvironment.get("stdgroup");
        PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();
        // Get the Discovery service for the current peer group.
        DiscoveryService discovery = currentGroup.getDiscoveryService();

        try
        {
            // Create an advertisement.
            PeerGroupAdvertisement advertisement =
                (PeerGroupAdvertisement)
                    AdvertisementFactory.newAdvertisement(
                        PeerGroupAdvertisement.getAdvertisementType());

            // Populate the various fields. For most of this, we'll create
            // our own values, but we'll need the Module Spec ID of our
            // current peer group.
            PeerGroupAdvertisement currentAdvertisement =
                currentGroup.getPeerGroupAdvertisement();

            // Set the values that must be unique for the new advertisement.
            advertisement.setName("NewGroup");
            advertisement.setDescription("PG for example4_6");
            advertisement.setPeerGroupID(IDFactory.newPeerGroupID());
```

```
            advertisement.setModuleSpecID(
                currentAdvertisement.getModuleSpecID());

            // Publish the advertisement locally.
            discovery.publish(advertisement, DiscoveryService.GROUP,
                10000, 1000);
            // Publish the advertisement remotely.
            discovery.remotePublish(advertisement,
                DiscoveryService.GROUP, 1000);
        }
        catch (IOException e)
        {
            println("Error publishing the advertisement to cache." + e);
            result = ShellApp.appMiscError;
        }
        return result;
    }
}
```

Not all the values for the newly created Peer Group Advertisement are exact copies; most important, the identifier for the peer group must be a new, unique ID. Creating a new ID is achieved using the `net.jxta.id.IDFactory` to create a new Peer Group ID:

```
advertisement.setPeerGroupID(IDFactory.newPeerGroupID());
```

The `IDFactory` class generates a unique identifier for a variety of advertisements that require a unique identifier, including peers, peer groups, pipes, and services.

One other ID that is added to the new Peer Group Advertisement is a Module Specification ID:

```
advertisement.setModuleSpecID(currentAdvertisement.getModuleSpecID());
```

This ID uniquely identifies a Module Specification Advertisement, which defines the set of services provided by the peer group. For this example, you simply copy the value, thereby

associating your Peer Group Advertisement with the same Module Specification Advertisement as the current group.

When you explore services and peer groups in Chapter 10, "Peer Groups and Services," you learn how to create a new Module Specification Advertisement and use it to create a new peer group and start the group's services. It's important to note that this example only publishes the new peer group's advertisement but does not actually start the new peer group's services.

To try out the `example4_6` command, start the Shell and flush the cache of Peer Group Advertisements:

```
JXTA>groups -f
```

After flushing the cached Peer Group Advertisements, executing the `groups` command again should result in an empty list. Implicitly, the peer is still aware of the default `NetPeerGroup`, and executing the `example4_6` command clones that group's advertisement and publishes the resulting advertisement both locally and remotely:

```
JXTA>example4_6
```

Another call to groups should display the newly published group:

```
JXTA>groups
group0: name = NewGroup
```

The `example4_6` command sets the local lifetime to 10 seconds (10,000 milliseconds) when it publishes the advertisement locally:

```
discovery.publish(advertisement, Discovery.GROUP,
    10000, 1000);
```

After 10 seconds, the Cache Manager clears the advertisement from the cache. Executing the `groups` command again returns an empty list, as expected.

Summary

This chapter demonstrated how the JXTA platform manages peer discovery and how the Java reference implementation provides a developer with the capability to send Discovery Query Messages to other peers and process the Discovery Responses Messages sent in response to queries.

In addition to performing discovery, the `DiscoveryService` interface and the implementation provided by the Java reference implementation can be used to publish advertisements to both the local cache and remote peers.

In the next chapter, you explore the Peer Resolver Protocol and the Resolver service. The Peer Resolver Protocol allows a peer to process and respond to generic queries. As you'll see, the Peer Resolver Protocol and the Resolver service provide the Discovery service with the capability to send queries to remote peers, process queries from other peers, and send responses to queries.

Chapter 5. The Peer Resolver Protocol

Chapter 4, "The Peer Discovery Protocol," showed how to discover peers and advertisements using the Discovery service, but it did not address how the Discovery service handles sending and receiving messages. The Discovery service isn't responsible for sending its Discovery Query and Response Messages. Instead, the Discovery service is built on top of another service, the Resolver service, which handles sending and receiving messages for the Discovery service.

This chapter details the Peer Resolver Protocol (PRP) and the Resolver service that implements the protocol. The PRP defines a protocol for sending a generic query to a named handler located on another peer and processing a generic response to a query. Other services in JXTA, such as the Discovery service, build on the capabilities of the Resolver service and the PRP to provide higher-level services.

Introducing the Peer Resolver Protocol

The Discovery service detailed in Chapter 4 described two types of messages: one for sending a discovery query and another for sending a response to a discovery query. The deceptive simplicity of the discovery messages hides several layers of abstraction that insulate the developer from the inner complexities of the JXTA protocols. To develop new solutions built on top of the JXTA platform, it's essential to understand these layers.

When a peer sends a Discovery Query Message using the `getRemoteAdvertisements` method, the `DiscoveryService` implementation doesn't simply create a Discovery Query Message and pass it over the network itself. Instead, the Discovery service uses another service, the Resolver service, to handle the details of sending the message on its behalf.

The Resolver service provides an implementation of the PRP, which defines how peers can exchange query and response messages.

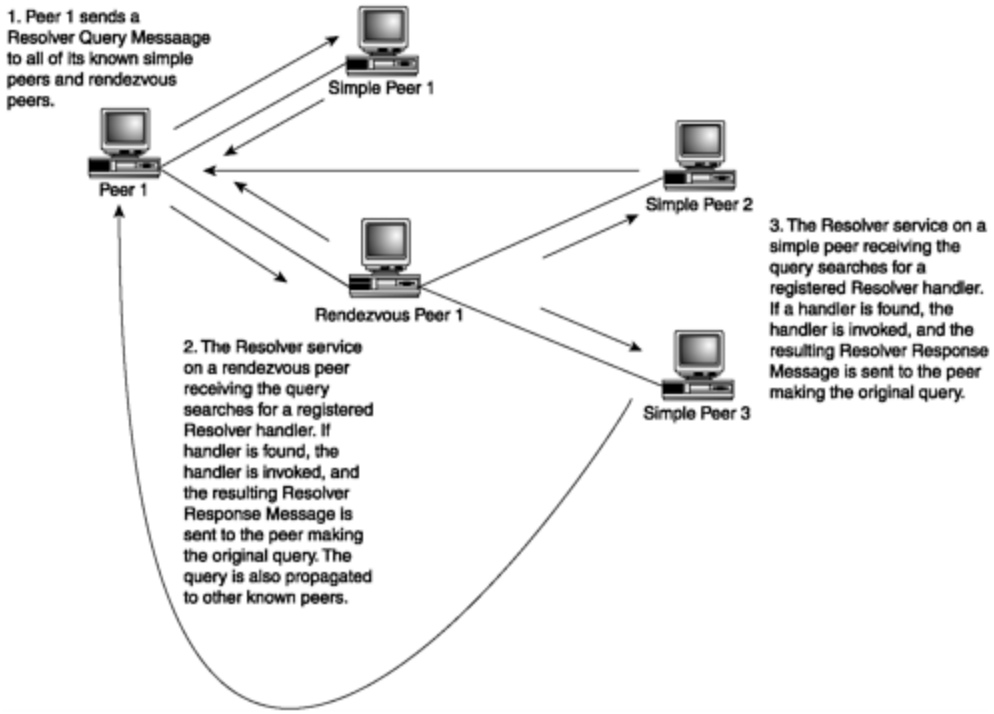
The Resolver service is responsible for wrapping a query string in a more generic message format and sending it to a specific handler on a remote peer. In the case of the Discovery service, the query string is the Discovery Query Message and the handler is the remote peer's Discovery service. On the remote peer, a Resolver service instance is responsible for passing an incoming message to the appropriate handler and sending any response generated by the handler.

In the general case, the Resolver service needs only two types of messages:

- **Resolver Query Message—** A message format for sending queries
- **Resolver Response Message—** A message format for sending responses to queries

These two message formats define generic messages to send queries and responses between peers, as shown in Figure 5.1. At each end, a handler registered with a peer group's Resolver service instance processes query strings and generates response strings.

Figure 5.1. Exchange of Resolver messages.



Like the Peer Discovery Protocol, a query is sent to known peers and propagated through known rendezvous peers. Any peer's Resolver service that receives a Resolver Query Message attempts to find a registered handler for the query. If a matching handler is found, the Resolver passes it the message and manages sending the response message generated by the handler.

Like the Discovery service, the Resolver service does not require a response to a Resolver Query Message. The registered handler might not generate a response to a given query and can indicate to the Resolver service the reason that it has not generated a response. A handler may not generate a response because it has decided that the query is invalid in some way. In this case, the query is not propagated to other peers. A handler might also indicate that it wants to learn the response generated by other peers in response to the query. To accomplish this, the handler can ask the Resolver service to resend the query in a manner that will allow it to receive the response generated by other peers.

Although the PRP is the default resolver protocol used by the JXTA reference implementation, developers can provide their own custom resolver mechanism. A developer might want to provide his own resolver mechanism to incorporate additional

information that provides a better or more efficient service. This custom solution could be built independently of the PRP or could be built on top of the PRP.

The Resolver Query Message

Queries to other peers are wrapped inside a Resolver Query Message using the format shown in Listing 5.1.

Listing 5.1 Resolver Query Message

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:ResolverQuery xmlns:jxta="http://jxta.org">
  <HandlerName> . . . </HandlerName>
  <Credential> . . . </Credential>
  <QueryID> . . . </QueryID>
  <SrcPeerID> . . . </SrcPeerID>
  <Query> . . . </Query>
</jxta:ResolverQuery>
```

The elements in the Resolver Query Message provide all the details that a peer's Resolver service needs to match the query to a registered handler:

- **`HandlerName`**— A required element containing a unique name specifying the name of the handler that the Resolver service on the destination peer should invoke to process the query. Because the Resolver service provides service within the context of a peer group, a handler name must be unique within a peer group. Only one handler of a given name should be registered on a given peer in a peer group, and this assumption is enforced in the Java reference implementation. If a second handler is registered with the Resolver for a peer group using a duplicate handler name, the first handler registered with the Resolver service is removed.

- **`Credential`**— An optional element containing an authentication token that identifies the source peer and its authorization to send the query to the peer group.

- **`QueryID`**— An optional element containing a long integer, encoded as a string, that defines an identifier for the query. This identifier should be unique to the query.

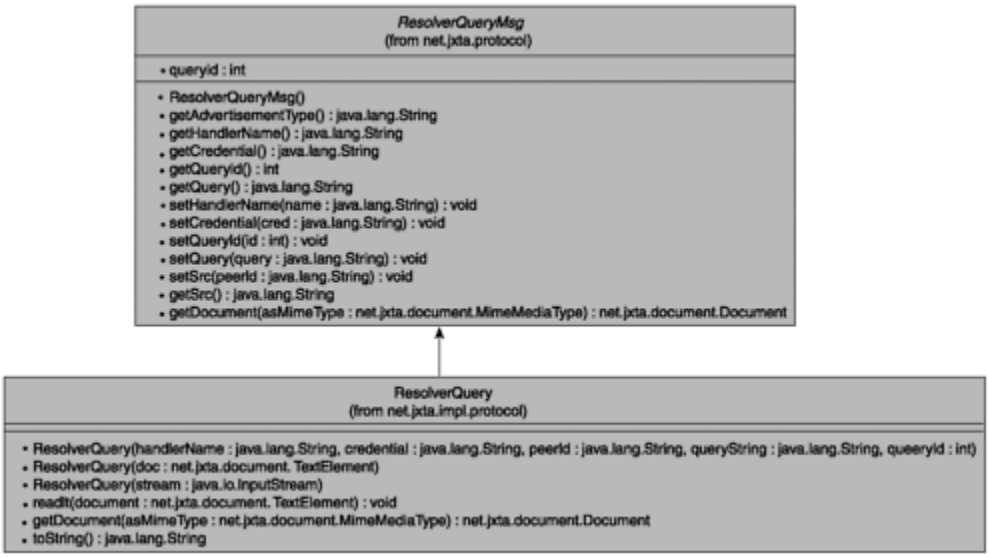
This identifier should be sent back in a response to this query, allowing the sender to match a response to a specific query.

- **`SrcPeerID`**— A required element containing the ID of the peer sending the query. This Peer ID uses the standard JXTA URN format, as described in the JXTA Protocols Specification.

- **`Query`**— A required element containing the query string being sent to the remote peer's handler. This string could be anything; it is the responsibility of the handler to understand how to parse this query string, process the query, and possibly generate a response message.

The implementation of the Resolver Query Message, as shown in Figure 5.2, is divided in a similar manner to the Discovery Query and Response Messages. The `ResolverQueryMsg` abstract class in the `net.jxta.protocol` package defines the basic interface, variables for the query's attributes, and accessors to manipulate the attributes. Only the `getDocument` method is abstract, allowing implementers to provide their own mechanism for rendering the query to a `Document` instance.

Figure 5.2. The Resolver Query Message classes.



The `ResolverQuery` class in the `net.jxta.impl.protocol` package provides an implementation of `getDocument` capable of creating a `StructuredTextDocument` representation of the query in the specified `MimeMediaType`. Several constructors can

create a `ResolverQuery` instance from a variety of input parameters, including an `InputStream`, or the raw query attributes.

A developer can create a Resolver Query Message at any time to send a query to a specific Resolver handler on a remote peer. For example, a call to `getRemoteAdvertisements` in the reference `DiscoveryService` implementation `DiscoveryServiceImpl` causes the `DiscoveryServiceImpl` to create a `DiscoveryQuery` instance, wrap it in a `ResolverQuery` instance, and send it using the Resolver service.

The Resolver Response Message

The Resolver Response Message responds to a Resolver Query Message using the format shown in Listing 5.2.

Listing 5.2 Resolver Response Message

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:ResolverResponse xmlns:jxta="http://jxta.org">
  <HandlerName> . . . </HandlerName>
  <Credential> . . . </Credential>
  <QueryID> . . . </QueryID>
  <Response> . . . </Response>
</jxta:ResolverResponse>
```

The Resolver Response Message provides similar details to the Resolver Query Message:

- **`HandlerName`**— A required element containing the name of a handler that should be invoked by the remote peer's Resolver service to process the response. A different handler name from that used in the query might be used to allow a different Resolver handler to process the response.

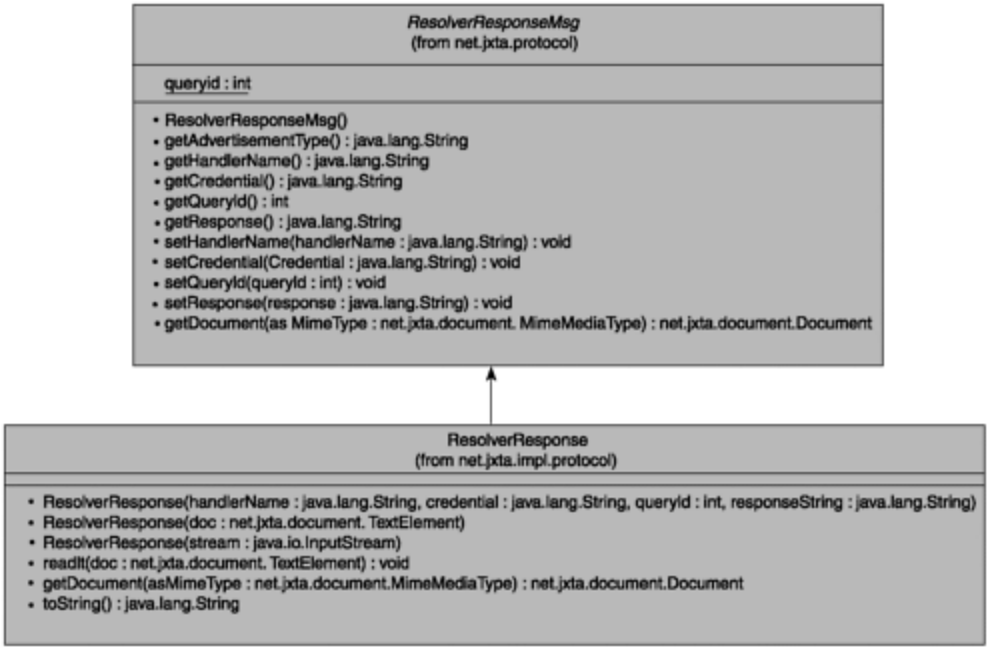
- **`Credential`**— An optional element containing an authentication token that identifies the peer sending the response and its authorization to send the response to the destination peer group.

- **`QueryID`**— An optional element containing a long integer, encoded as a string, that defines an identifier for the query. This identifier should correspond to the `QueryID` sent in the query that caused the peer to generate this Resolver Response Message. If the `QueryID` provided in the original query is unique to the query and the handler, the sender can match this Resolver Response Message to the original query. Matching the response to a given query might be necessary to provide useful functionality in a P2P application.

- **`Response`**— A required element containing the response string being sent to the remote peer's handler. This string could be anything; it is the responsibility of the handler to understand how to parse this response string.

In the Java reference implementation of JXTA, the abstract definition of the Resolver Response Message is defined by `ResolverResponseMsg` in the `net.jxta.protocol` package, as shown in Figure 5.3. The reference implementation of the abstract class is implemented by the `ResolverResponse` class in the `net.jxta.impl.protocol` package. ResolverResponseMsg

Figure 5.3. The Resolver Response Message classes.



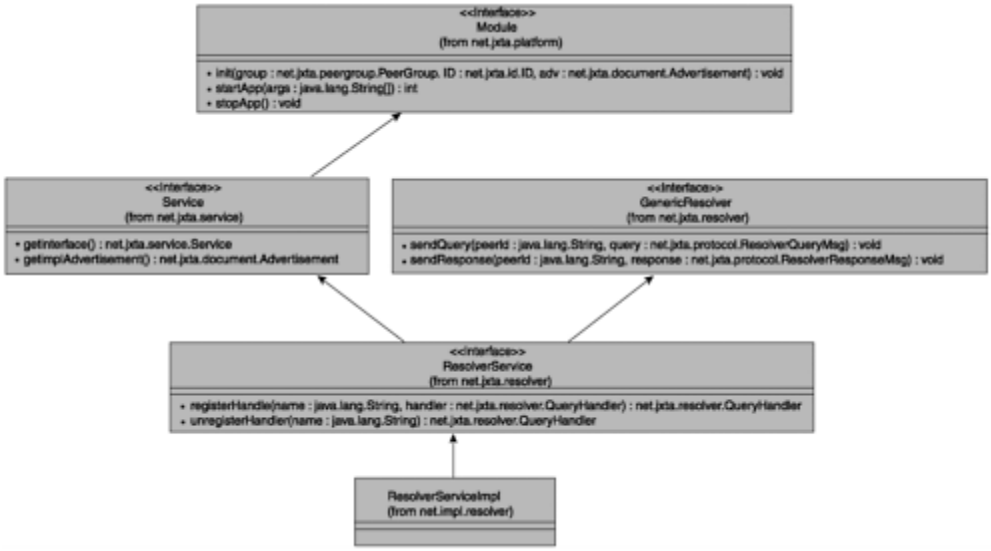
A Resolver Response Message can be used to "push" information to peers by sending a Resolver Response Message without first receiving a Resolver Query Message. This

capability is used by the `DiscoveryService` implementation `DiscoveryServiceImpl` to publish advertisements to remote peers whenever the `remotePublish` method is called.

The Resolver Service

The Resolver service, another JXTA core service, provides a simple interface that developers can use to send queries and responses between members of a peer group. The Resolver service is defined by the `ResolverService` interface in the `net.jxta.resolver` package, shown in Figure 5.4, which is derived from the `GenericResolver` interface.

Figure 5.4. The Resolver service interfaces and classes.

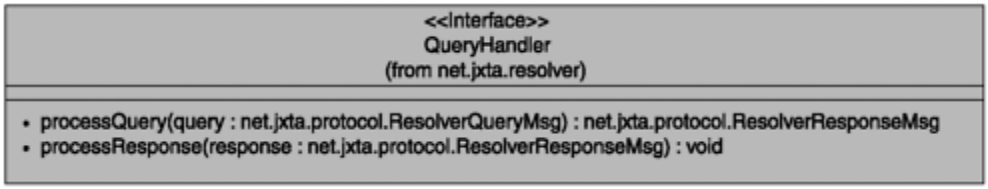


The `GenericResolver` interface defines the methods for sending queries and responses using implementations of `ResolverQueryMsg` and `ResolverResponseMsg`. More important, the `ResolverService` interface defines the methods for registering and unregistering an implementation of the `net.jxta.resolver.QueryHandler` interface and associating it with a handler name. A registered `QueryHandler` instance is invoked when the Resolver service receives a Resolver Query or Response Message whose `HandlerName` matches the handler string used to register the handler with the `ResolverService` for a peer group.

The *QueryHandler* Interface

The `QueryHandler` interface, shown in Figure 5.5, is similar to the `DiscoveryListener` interface in Chapter 4. Like `DiscoveryListener`, the `QueryHandler` interface provides a developer with a way to provide his own mechanism for handling response messages. Unlike `DiscoveryListener`, the `QueryHandler` interface also provides a developer with a mechanism for handling query messages received from other peers.

Figure 5.5. The `QueryHandler` interface.



To begin handling queries, a `QueryHandler` instance first must be registered with a peer group's `ResolverService` using a unique handler name. After it is registered, a peer group's `ResolverService` instance invokes the `QueryHandler` 's `processQuery` method to process `ResolverQueryMsg` instances addressed to the handler. The `processQuery` implementation is responsible for extracting and processing the query string from the `ResolverQueryMsg`. This processing can result in one of five outcomes:

- The `processQuery` method returns a populated `ResolverResponseMsg` object containing the response to the query. The `ResolverService` instance handles sending this response back to the peer that sent the original query.

- The `processQuery` method throws a `NoResponseException`, indicating that the handler has no response to the query. If the peer is a rendezvous peer, the ResolverService instance still propagates the query to other peers in the peer group.

- The `processQuery` method throws a `ResendQueryException`, indicating that the handler has no response to the query but would be interested in learning the response given by other peers. If the peer is a rendezvous peer, the `ResolverService` propagates the query message as usual to other peers in the peer group. In addition to propagating the message, the `ResolverService` instance

resends the message (masquerading as the source peer) to obtain the responses provided by other peers to the query.

- The `processQuery` method throws a `DiscardException`, indicating that the `ResolverService` should discard the query entirely. The `ResolverService` instance discards the query and does not propagate the query to other peers in the peer group. A query can be discarded because the query, despite being well formed, might be invalid in some way.

- The `processQuery` method throws an `IOException` when the handler cannot process the query, possibly due to an error in the format of the query string. In the reference `ResolverService` implementation, this exception causes the Resolver service to act in the same manner as when a `DiscardException` occurs.

The `QueryHandler` 's `processResponse` method is invoked by the `ResolverService` to process a `ResolverResponseMsg` instance. Unlike `processQuery`, `processResponse` doesn't produce any results or throw any exceptions. Either the response is processed or it isn't. The `ResolverService` instance doesn't need to know anything about the results of the processing.

One thing that might seem curious is that the Resolver service and the `QueryHandler` interface don't provide information on how the query or response strings are formatted. No mechanism exists for a peer to discover how to format a query string for a given handler or what format to expect in response to a successful query. The details of the query and response string formatting are implicit in the implementation of the handler, and JXTA does not provide any way of discovering how to invoke the handler. This is one area that JXTA does not address but that could be addressed in the future by adopting one of the forthcoming XML-based standards for service discovery, such as the Web Services Description Language (WSDL).

Implementing a Resolver Handler

The example covered in this section creates a simple handler that allows a peer to query a remote peer for the value of a specified base raised to a specified power. The query string provides the base and power values in the format shown in Listing 5.3.

Listing 5.3 The Example Query Message

```
<?xml version="1.0"?>
<example:ExampleQuery xmlns:example="http://jxta.org">
  <base> . . . </base>
  <power> . . . </power>
</example:ExampleQuery>
```

Responses to the query provide the answer to the query using the format in Listing 5.4.

Listing 5.4 The Example Response Message

```
<?xml version="1.0"?>
<example:ExampleResponse xmlns:example="http://jxta.org">
  <base> . . . </base>
  <power> . . . </power>
  <answer> . . . </answer>
</example:ExampleResponse>
```

The example Resolver handler accepts a query, extracts the base and power values, calculates the value of the base raised to the power, and returns a response message populated with the base, power, and answer values.

Creating the Query and Response Strings

Implementing a Resolver handler requires a developer only to provide an implementation of the `QueryHandler` interface and register the handler with a peer group's Resolver service. However, a developer should still abstract the process of parsing query strings and formatting response strings, in the interest of readability and maintainability.

The Discovery service, covered in Chapter 4, relies on the `DiscoveryQueryMsg`, `DiscoveryQuery`, `DiscoveryResponseMsg`,and `DiscoveryResponse` classes. These classes provided a mechanism for the `DiscoveryService` implementation to produce or consume a query or response string in a fairly abstract fashion. For this example, there's no need to go as far as defining both an abstract class and an implementation for the query and response objects. The query and response objects used in this example use a similar

approach to provide encapsulated parsing and formatting functionality. Listing 5.5 shows the source code for an object to handle parsing and formatting the query XML shown in Listing 5.4.

Listing 5.5 Source Code for *ExampleQueryMsg.java*

```
package net.jxta.impl.shell.bin.example5_1;

import java.io.InputStream;
import java.io.IOException;
import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.Document;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;


/**
 * An example query message, which will be wrapped by a Resolver Query
 * Message to send the query to other peers. The query essentially asks
 * a simple math question: "What is the value of (base) raised to (power)?"
*/
public class ExampleQueryMsg
{
    /**
     * The base for query.
     */
    private double base = 0.0;

    /**
```

```
 * The power for the query.
 */
private double power = 0.0;


/**
 * Creates a query object using the given base and power.
 *
 * @param   aBase the base for the query.
 * @param   aPower the power for the query.
 */
public ExampleQueryMsg(double aBase, double aPower)
{
    super();

    this.base = aBase;
    this.power = aPower;
}

/**
 * Creates a query object by parsing the given input stream.
 *
 * @param       stream the InputStream source of the query data.
 * @exception   IOException if an error occurs reading the stream.
 */
public ExampleQueryMsg(InputStream stream) throws IOException
{
        StructuredTextDocument document = (StructuredTextDocument)
            StructuredDocumentFactory.newStructuredDocument(
                new MimeMediaType("text/xml"), stream);

    Enumeration elements = document.getChildren();

    while (elements.hasMoreElements())
    {
        TextElement element = (TextElement) elements.nextElement();
```

```
            if(element.getName().equals("base"))
            {
                base =
Double.valueOf(element.getTextValue()).doubleValue();
                continue;
            }

            if(element.getName().equals("power"))
            {
                power = Double.valueOf(
                    element.getTextValue()).doubleValue();
                continue;
            }
        }
    }

    /**
     * Returns the base for the query.
     *
     * @return  the base value for the query.
     */
    public double getBase()
    {
        return base;
    }

    /**
     * Returns a Document representation of the query.
     *
     * @param   asMimeType the desired MIME type representation for the
     *          query.
     * @return  a Document form of the query in the specified MIME
     *          representation.
     */
    public Document getDocument(MimeMediaType asMimeType)
```

```
{
    StructuredDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            asMimeType, "example:ExampleQuery");
    Element element;

    element = document.createElement(
        "base", Double.toString(getBase()));
    document.appendChild(element);

    element = document.createElement("power",
        Double.toString(getPower()));
    document.appendChild(element);

    return document;
}

/**
 * Returns the power for the query.
 *
 * @return  the power value for the query.
 */
public double getPower()
{
    return power;
}

/**
 * Returns an XML String representation of the query.
 *
 * @return  the XML String representing this query.
 */
public String toString()
{
    try
    {
```

```
            StringWriter out = new StringWriter();
            StructuredTextDocument doc = (StructuredTextDocument)
                getDocument(new MimeMediaType("text/xml"));
            doc.sendToWriter(out);
            return out.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

Like the `DiscoveryQueryMsg` and `DiscoveryQuery` classes, the `ExampleQueryMsg` class defines fields for the query, methods for rendering the message as a `Document` and a `String`, and constructors for populating a query. The `getDocument` method creates a `Document` for the given `MimeMediaType` using the `StructuredDocumentFactory` class and adds the `base` and `power` fields. The `toString` method provides a convenient way to render the query object to an XML string, resulting in a query string in the format defined at the beginning of this section.

The encapsulation of the response message format is almost identical, as shown in Listing 5.6.

Listing 5.6 Source Code for *ExampleResponseMsg.java*

```
package net.jxta.impl.shell.bin.example5_1;

import java.io.InputStream;
import java.io.IOException;
import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.Document;
import net.jxta.document.Element;
```

```
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;


/**
 * An example query response, which will be wrapped by a Resolver Response
 * Message to send the response to the query. The response contains the
 * answer to the simple math question posed by the query.
 */
public class ExampleResponseMsg
{
    /**
     * The base from the original query.
     */
    private double base = 0.0;

    /**
     * The power from the original query.
     */
    private double power = 0.0;

    /**
     * The answer value for the response.
     */
    private double answer = 0;


    /**
     * Creates a response object using the given answer value.
     *
     * @param   anAnswer the answer for the response.
     */
```

```
    public ExampleResponseMsg(double aBase, double aPower, double
anAnswer)
    {
        this.base = aBase;
        this.power = aPower;
        this.answer = anAnswer;
    }

    /**
     * Creates a response object by parsing the given input stream.
     *
     * @param       stream the InputStream source of the response data.
     * @exception   IOException if an error occurs reading the stream.
     */
    public ExampleResponseMsg(InputStream stream) throws IOException
    {
        StructuredTextDocument document = (StructuredTextDocument)
            StructuredDocumentFactory.newStructuredDocument(
                new MimeMediaType("text/xml"), stream);

        Enumeration elements = document.getChildren();

        while (elements.hasMoreElements())
        {
            TextElement element = (TextElement) elements.nextElement();

            if(element.getName().equals("answer"))
            {
                answer = Double.valueOf(
                    element.getTextValue()).doubleValue();
                continue;
            }

            if(element.getName().equals("base"))
            {
```

```
                base =
Double.valueOf(element.getTextValue()).doubleValue();
                continue;
            }

            if(element.getName().equals("power"))
            {
                power = Double.valueOf(
                    element.getTextValue()).doubleValue();
                continue;
            }
        }
    }

    /**
     * Returns the answer for the response.
     *
     * @return  the answer value for the response.
     */
    public double getAnswer()
    {
        return answer;
    }

    /**
     * Returns the base for the query.
     *
     * @return  the base value for the query.
     */
    public double getBase()
    {
        return base;
    }

    /**
     * Returns a Document representation of the response.
```

```
 *
 * @param   asMimeType the desired MIME type representation for
 *          the response.
 * @return  a Document form of the response in the specified MIME
 *    representation.
 */
public Document getDocument(MimeMediaType asMimeType)
{
    Element element;
    StructuredDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            asMimeType, "example:ExampleResponse");

    element = document.createElement(
        "base", Double.toString(getBase()));
    document.appendChild(element);

    element = document.createElement("power",
        Double.toString(getPower()));
    document.appendChild(element);

    element = document.createElement("answer",
        (new Double(getAnswer()).toString()));
    document.appendChild(element);

    return document;
}
/**
 * Returns the power for the query.
 *
 * @return  the power value for the query.
 */
public double getPower()
{
    return power;
}
```

```
    /**
     * Returns an XML String representation of the response.
     *
     * @return  the XML String representing this response.
     */
    public String toString()
    {
        try
        {
            StringWriter buffer = new StringWriter();
            StructuredTextDocument document = (StructuredTextDocument)
                getDocument(new MimeMediaType("text/xml"));
            document.sendToWriter(buffer);

            return buffer.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

These objects simplify the task of creating a Resolver Query or Response Message. For example, a developer can create a query string and wrap it in a Resolver Query Message using only this code:

```
ExampleQueryMsg equery =
    new ExampleQueryMsg(base, power);
ResolverQuery query = new ResolverQuery("ExampleHandler",
    "JXTACRED", localPeerId, equery.toString(), queryId);
```

The query string can be extracted and parsed using this code:

```
equery = new ExampleQueryMsg(
    new ByteArrayInputStream((query.getQuery()).getBytes()));
```

Both of these examples demonstrate how much simpler it is to use the encapsulated query and response objects compared to the alternative of manually formatting or parsing the query or response string.

Implementing the *QueryHandler* Interface

The task of implementing the `QueryHandler` interface is greatly simplified by the query and response objects defined in the previous section. Listing 5.7 shows the source code for the sample `QueryHandler`.

Listing 5.7 Source Code for *ExampleHandler.java*

```
package net.jxta.impl.shell.bin.example5_1;

import java.io.ByteArrayInputStream;
import java.io.IOException;

import net.jxta.exception.NoResponseException;
import net.jxta.exception.DiscardQueryException;
import net.jxta.exception.ResendQueryException;

import net.jxta.impl.protocol.ResolverResponse;

import net.jxta.protocol.ResolverQueryMsg;
import net.jxta.protocol.ResolverResponseMsg;

import net.jxta.resolver.QueryHandler;



/**
 * A simple handler to process Resolver queries.
 */
class ExampleHandler implements QueryHandler
```

```
{
    /**
     * Processes the Resolver query message and returns a response.
     *
     * @param       query the Resolver Query Message to be processed.
     * @return      a Resolver Response Message to send to the peer
     *              responsible for sending the query.
     * @exception   IOException throw if the query string can't be parsed.
     */
    public ResolverResponseMsg processQuery(ResolverQueryMsg query)
        throws IOException, NoResponseException, DiscardQueryException,
            ResendQueryException
    {
        ResolverResponse response;
        ExampleQueryMsg eq;
        double answer = 0.0;

        System.out.println("Processing query...");

        // Parse the message from the query string.
        eq = new ExampleQueryMsg(
            new ByteArrayInputStream((query.getQuery()).getBytes()));

        // Perform the calculation.
        answer = Math.pow(eq.getBase(), eq.getPower());

        // Create the response message.
        ExampleResponseMsg er = new ExampleResponseMsg(eq.getBase(),
            eq.getPower(), answer);

        // Wrap the response message in a resolver response message.
        response = new ResolverResponse("ExampleHandler",
            "JXTACRED", query.getQueryId(), er.toString());

        return response;
    }
```

```
    /**
     * Process a Resolver response message.
     *
     * @param   response the Resolver Response Message to be processed.
     */
    public void processResponse(ResolverResponseMsg response)
    {
        ExampleResponseMsg er;

        System.out.println("Processing response...");

        try
        {
            // Extract the message from the Resolver response.
            er = new ExampleResponseMsg(
                new ByteArrayInputStream(
                    (response.getResponse()).getBytes()));

            // Print out the answer given in the response.
            System.out.println(
                "The value of " + er.getBase() + " raised to "
                    + er.getPower() + " is: " + er.getAnswer());
        }
        catch (IOException e)
        {
            // This is not the right type of response message, or
            // the message is improperly formed. Ignore the message,
            // do nothing.
        }
    }
}
```

The `QueryHandler` interface's only two methods, `processQuery` and `processResponse`, use the objects defined in the previous section to parse and format the query and response

strings. The only real work that is done by the `ExampleHandler` is the calculation of the response's `answer` value using the query's `base` and `power` values.

Registering the Handler with the *ResolverService* Instance

To see the `QueryHandler` implementation created in the previous section in action, an instance of `ExampleHandler` needs to be registered with a peer group's `ResolverService` instance. The following example `shell` command, shown in Listing 5.8, registers an `ExampleHandler` instance with the current peer group's `ResolverService` instance and sends an `ExampleQueryMsg` using input values provided by the user.

Listing 5.8 Source Code for *example5_1.java*

```
package net.jxta.impl.shell.bin.example5_1;

import net.jxta.impl.protocol.ResolverQuery;

import net.jxta.impl.shell.GetOpt;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;

import net.jxta.peergroup.PeerGroup;

import net.jxta.resolver.QueryHandler;
import net.jxta.resolver.ResolverService;



/**
 * A simple application to demonstrate the use of the Resolver service
 * to register a QueryHandler instance and process queries.
 */
public class example5_1 extends ShellApp
{
    /**
```

```
 * A flag indicating if the example's handler should be unregistered
 * from the peer group's Resolver service.
 */
private boolean removeHandler = false;

/**
 * A name to use to register the example handler with the
 * Resolver service.
 */
private String name = "ExampleHandler";

/**
 * The base value for the query.
 */
private double base = 0.0;

/**
 * The power value for the query.
 */
private double power = 0.0;


/**
 * Manages adding or removing the handler from the Resolver service.
 *
 * @param   resolver the Resolver service with which to register or
 *          unregister a handler.
 */
private void manageHandler(ResolverService resolver)
{
    if (removeHandler)
    {
        // Unregister the handler from the Resolver service.
        ExampleHandler handler =
            (ExampleHandler) resolver.unregisterHandler(name);
    }
```

```
    else
    {
        // Create a new handler.
        ExampleHandler handler = new ExampleHandler();

        // Register the handler with the Resolver service.
        resolver.registerHandler(name, handler);
    }
}

/**
 * Parses the command-line arguments and initializes the command
 *
 * @param       args the arguments to be parsed.
 * @exception   IllegalArgumentException if an invalid parameter
 *              is passed.
 */
private void parseArguments(String[] args)
    throws IllegalArgumentException
{
    int option;

    // Parse the arguments to the command.
    GetOpt parser = new GetOpt(args, "b:p:r");

    while ((option = parser.getNextOption()) != -1)
    {
        switch (option)
        {
            case 'b' :
            {
                try
                {
                    // Obtain the "base" element for the query.
                    base = (new Double(
                        parser.getOptionArg())).doubleValue();
```

```
                }
                catch (Exception e)
                {
                    // Default to 0.0
                    base = 0.0;
                }

                break;
            }

            case 'p' :
            {
                try
                {
                    // Obtain the "power" element for the query.
                    power = (new Double(
                        parser.getOptionArg())).doubleValue();
                }
                catch (Exception e)
                {
                    // Default to 0.0
                    power = 0.0;
                }

                break;
            }

            case 'r' :
            {
                // Remove the handler.
                removeHandler = true;
                break;
            }
        }
    }
}
```

```
/**
 * The implementation of the Shell command, invoked when the command
 * is started by the user from the Shell.
 *
 * @param   args the command-line arguments passed to the command.
 * @return  a status code indicating the success or failure of
 *          the command.
 */
public int startApp(String[] args)
{
    int result = appNoError;

    // Get the shell's environment.
    ShellEnv theEnvironment = getEnv();

    // Use the environment to obtain the current peer group.
    ShellObject theShellObject = theEnvironment.get("stdgroup");
    PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

    // Get the Resolver service for the current peer group.
    ResolverService resolver = currentGroup.getResolverService();

    try
    {
        // Parse the command-line arguments.
        parseArguments(args);
    }
    catch (IllegalArgumentException e)
    {
        println("Incorrect parameters passed to the command.");
        result = ShellApp.appParamError;
    }

    // Manage the handler for the Resolver service. This
    // adds or removes the handler as specified by the
    // command-line parameters.
```

```
        manageHandler(resolver);

        // If we're not removing the handler, send a query using
        // the Resolver service.
if (!removeHandler)
        {
            ExampleQueryMsg equery = new ExampleQueryMsg(base, power);
            String localPeerId = currentGroup.getPeerID().toString();

            // Wrap the query in a resolver query.
            ResolverQuery query = new ResolverQuery("ExampleHandler",
                "JXTACRED", localPeerId, equery.toString(), 0);

            // Send the query using the resolver.
            println("Sending base="+base+", power="+power);
            resolver.sendQuery(null, query);
        }

        return result;
    }
}
```

Of particular importance is the registration of the `ExampleHandler`:

```
// Register the handler with the Resolver service.
resolver.registerHandler(name, handler);
```

The `name` variable defines the name of the handler that identifies this handler to the `ResolverService` instance. In the `example5_1` command, `name` is set to `ExampleHandler`. A Resolver Query Message or a Resolver Response Message must use the same handler name to identify the target handler for its query or response string.

Because a peer group's `ResolverService` instance can define only one handler with a given name, the `registerHandler` method replaces an existing handler. Any handler

previously registered with the `ResolverService` instance using the same handler name is returned by the `registerHandler` method.

Sending a Resolver Query Message

To send a query, the `example5_1` command creates an `ExampleQueryMsg` object using the `base` and `power` values provided by the user and wraps it in a `ResolverQuery` object:

```
ExampleQueryMsg equery = new ExampleQueryMsg(base, power);
String localPeerId = currentGroup.getPeerID().toString();
ResolverQuery query = new ResolverQuery("ExampleHandler",
    "JXTACRED", localPeerId, equery.toString(), 0);
```

The identifier for the local peer, `localPeerId`, is retrieved from the `PeerGroup` object holding the current peer group in the Shell when the command is invoked. The `JXTACRED string` provides a value for the `Credential` in the `ResolverQuery`. Currently, the JXTA reference implementation doesn't provide any abstract mechanism for validating credentials, although this feature is expected in the future. Currently, developers can implement their own credential validation schemes within their `QueryHandler` implementations until this shortcoming is addressed.

Finally, the Resolver Query Message is sent to all peers in the `ResolverService` instance's peer group using this line:

```
resolver.sendQuery(null, query);
```

The first parameter identifies the Peer ID for the destination of the query. If this parameter is `null`, the `ResolverService` instance propagates the query to all peers in the peer group.

When a Resolver service receives a Resolver Query Message, it extracts the `HandlerName`, checks for a matching registered `QueryHandler` instance, and, if one exists, passes the Resolver Query Message object to the handler's `processQuery` method.

Using the *ExampleHandler* Class

To see the example in action, two peers on the P2P network must register an `ExampleHandler` instance with a specific peer group's `ResolverService` instance using the same handler name. Because it's unlikely that another peer will be running the example code at the same time, you must start two instances of the Shell. To start two instances of the Shell, follow these steps:

1. Delete the `PlatformConfig` file and the `pse` and `cm` directories from your `Shell` directory. Run the `Shell`, force reconfiguration using the `peerconfig` command, and `exit` the `Shell`.

2. Copy the `Shell` subdirectory from the JXTA Demo install directory into a directory called `Shell2`. This directory should be at the same directory level as the original `Shell` subdirectory.

3. Compile the example's code, and place a copy in both the `Shell` and `Shell2` subdirectories. This is required because the example code must be to be available to both `Shell` instances.

4. Run the `Shell` in the `Shell` directory from the command line, as in previous examples. Configure it as usual.

5. Run the `Shell` in the `Shell2` directory from the command line, as in previous examples. In the TCP Settings section of the Advanced tab, specify a different TCP port number (for example, 9702). In the HTTP Settings section of the Advanced tab, specify a different HTTP port number (for example, 9703). In the Basic tab, enter a different name for the peer.

After each Shell has loaded, issue a peer discovery in each Shell using `peers -r`, and ensure that each peer can see the other using the `peers` command. Each peer must be capable of seeing the other peer's name in the list returned by `peers` for the example to work. When both peers can see each other, run the example in the first Shell instance:

```
JXTA>example5_1
```

The command registers an `ExampleHandler` with the current peer group's Resolver service and sends a default query. The default query for the example uses a value of `0.0` for both the `base` and the `power` attributes. No response to this query is received because probably no other peer on the system at this time has a matching handler registered with its Resolver service for the current peer group.

Run the example in the second Shell instance to register a handler. This time, the default query is handled by the `ExampleHandler` registered in the first Shell instance. The first Shell's `ExampleHandler` instance prints to the command console (not the Shell console):

```
Processing query...
```

This indicates that the Resolver service has received a query and passed it to the `processQuery` method of the `ExampleHandler`. The `ExampleHandler` 's `processQuery` method has been invoked correctly, and the handler is processing the query. When the handler returns a response, the Resolver service sends it back to the second Shell instance's peer. When this response is received by the second Shell instance, `ExampleHandler` prints the results to the command console (again, not to the Shell console):

```
Processing response...
The value of 0.0 raised to the power 0.0 is: 1.0
```

This indicates that the `processResponse` method of the `ExampleHandler` registered in the second Shell has been invoked by the Resolver service correctly. Now that both peers have registered a handler, try sending a more meaningful query using this line:

```
JXTA>example5_1 -b4 -p2
```

The query asks other peer for the value of `4` raised to the power 2. The other peer should respond with the value `16`.

Unregistering the Handler

When an application no longer wants a handler to receive messages, it can unregister the handler from the Resolver service. To unregister the handler, the `unregister` method is called using the name originally used to register to handler:

```
ExampleHandler handler = (ExampleHandler)
    resolver.unregisterHandler(name);
```

Unregistering the handler returns the `QueryHandler` instance that the `ResolverService` has unregistered. If the call to `unregister` returns `null`, the `ResolverService` instance cannot find any registered handler instance with the given name.

Sending Responses

The `example4_6` command developed in the Chapter 4 showed how the Discovery service can be used to publish advertisements to other peers using the `remotePublish` method. To do this, the Discovery service sends a Discovery Response Message using the `ResolverService`'s `sendResponse` method:

```
public void sendResponse(String destPeer, ResolverResponseMsg response);
```

The `sendResponse` method allows a peer to send a Resolver Response Message without first receiving a Resolver Query Message. Using this method, the example given in Listing 5.9 allows a peer to publish answers to other peers.

Listing 5.9 Source Code for *example5_2.java*

```
package net.jxta.impl.shell.bin.example5_2;

import net.jxta.impl.protocol.ResolverResponse;

import net.jxta.impl.shell.GetOpt;
```

```
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;

import net.jxta.impl.shell.bin.example5_1.ExampleResponseMsg;

import net.jxta.peergroup.PeerGroup;

import net.jxta.resolver.ResolverService;


/**
 * A simple application to demonstrate the use of the Resolver service to
 * send a Resolver Response Message without first receiving a Resolver
 * Query Message.
 */
public class example5_2 extends ShellApp
{
    /**
     * The base value for the response.
     */
    private double base = 0.0;

    /**
     * The power value for the response.
     */
    private double power = 0.0;

    /**
     * The answer value for the response.
     */
    private double answer = 0;


    /**
     * Parses the command-line arguments and initializes the command
```

```
 *
 * @param      args the arguments to be parsed.
 * @exception   IllegalArgumentException if an invalid parameter
 *            is passed.
 */
private void parseArguments(String[] args)
    throws IllegalArgumentException
{
    int option;

    // Parse the arguments to the command.
    GetOpt parser = new GetOpt(args, "b:p:a:");

    while ((option = parser.getNextOption()) != -1)
    {
        switch (option)
        {
            case 'b' :
            {
                try
                {
                    // Obtain the "base" element for the response.
                    base = (new Double(
                        parser.getOptionArg())).doubleValue();
                }
                catch (Exception e)
                {
                    // Default to 0.0
                    base = 0.0;
                }

                break;
            }

            case 'p' :
            {
```

```
                try
                {
                    // Obtain the "power" element for the response.
                    power = (new Double(
                        parser.getOptionArg())).doubleValue();
                }
                catch (Exception e)
                {
                    // Default to 0.0
                    power = 0.0;
                }

                break;
            }

            case 'a' :
            {
                try
                {
                    // Obtain the "answer" element for the response.
                    answer =  (new Double(
                        parser.getOptionArg())).doubleValue();
                }
                catch (Exception e)
                {
                    // Default to 0.0
                    answer = 0.0;
                }

                break;
            }
        }
    }
}

/**
```

```
 * The implementation of the Shell command, invoked when the command
 * is started by the user from the Shell.
 *
 * @param   args the command-line arguments passed to the command.
 * @return  a status code indicating the success or failure of
 *          the command.
 */
public int startApp(String[] args)
{
    int result = appNoError;

    // Get the shell's environment.
    ShellEnv theEnvironment = getEnv();

    // Use the environment to obtain the current peer group.
    ShellObject theShellObject = theEnvironment.get("stdgroup");
    PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

    // Get the Resolver service for the current peer group.
    ResolverService resolver = currentGroup.getResolverService();

    try
    {
        // Parse the command-line arguments.
        parseArguments(args);
    }
    catch (IllegalArgumentException e)
    {
        println("Incorrect parameters passed to the command.");
        return ShellApp.appParamError;
    }

    String localPeerId = currentGroup.getPeerID().toString();
    ExampleResponseMsg eresponse =
        new ExampleResponseMsg(base, power, answer);
    ResolverResponse pushRes = new ResolverResponse("ExampleHandler",
```

```
            "JXTACRED", 0, eresponse.toString());

        // Print out the information we're about to send.
        System.out.println(
            "Sending: base=" + base + ", power=" + power
                + ", answer=" + answer);

        // Send the response using the resolver.
        resolver.sendResponse(null, pushRes);

        return result;
    }
}
```

A Resolver Response Message is created by the command in a similar fashion to the `ExampleHandler`'s `processQuery` method in the previous example:

```
ExampleResponseMsg eresponse =
    new ExampleResponseMsg(base, power, answer);
ResolverResponse pushRes = new ResolverResponse("ExampleHandler",
    "JXTACRED", 0, eresponse.toString());
```

Using the arguments passed to the command, the `example5_2` command wraps an `ExampleResponseMsg` in a `ResolverResponse` message. Unlike the previous example, the response is sent using the `ResolverService` directly:

```
resolver.sendResponse(null, pushRes);
```

The first parameter to the `sendResponse` method specifies a destination peer, in the form of a Peer ID `String`. If this string is `null`, the `ResolverService` instance sends the response message to every known peer and propagates the message via known rendezvous peers.

To test the example, start two Shell instances using the procedure given in the previous example. Register an `ExampleHandler` in each instance using the `example5_1` command and then invoke the `example5_2` command in the first Shell instance using this line:

```
JXTA>example5_2 -b4 -p2 -a16
```

This command sends an `ExampleResponseMsg` to all known peers, using a base value of `4`, a power value of `2`, and an answer value of `16`. The second Shell instance receives the message, and the Resolver service invokes the `ExampleHandler` to print a message to the system:

```
The value of 4.0 raised to the power 2.0 is: 16.0
```

The `example5_2` command enables a user to send a response without requiring a query first, allowing a peer to publish an answer before the question has been asked. One of the interesting things to note here is that a peer can provide incorrect answers! This is actually a core problem in P2P computing that is currently the subject of much discussion.

Summary

In this chapter, you learned that the Resolver service is used as a building block by the Discovery service to provide a more generic message-handling framework. Using the Resolver service, you should now be able to create and register handlers to provide your own functionality to a peer group.

In the next chapter, you explore the Rendezvous Protocol and the Rendezvous service. Despite its name, the Rendezvous service is not solely used to provide rendezvous peer services to other peers. The Rendezvous service is a building block that can also be used by services on a peer to propagate messages to other peers within the same peer group. For example, the Resolver service explored in this chapter used the Rendezvous service to propagate queries to remote peers. The next chapter details the protocol behind the Rendezvous service and how it can be used by developers to handle propagating messages to other peers.

Chapter 6. The Rendezvous Protocol

In Chapter 5, "The Peer Resolver Protocol," you learned that the Resolver service provides the foundation for the Discovery's service's capability to query remote peers and respond to queries from remote peers. Just as the Discovery service relies on the capabilities of the Resolver service, the Resolver service relies on the capabilities of another service: the Rendezvous service. The Rendezvous service is responsible not only for allowing a user to propagate messages to other peers via a rendezvous peer, but also for providing rendezvous peer services to other peers on the network.

This chapter explains the Rendezvous Protocol (RVP) that simple peers use to connect to rendezvous peers to propagate messages to other peers on their behalf. As you'll see, the Rendezvous service implementation of the RVP has a dual role, providing a unified API for propagating messages, independent of whether a peer is configured to act as a rendezvous peer.

Introducing the Rendezvous Protocol

Chapter 2, "P2P Concepts," introduced the concept of a rendezvous peer, a peer used to propagate messages within a peer group on another peer's behalf. In JXTA, a rendezvous peer provides simple peers in private networks with the capability to broadcast messages to other members of a peer group outside the private network. This functionality is independent of the underlying network transport, allowing message propagation over transports that don't support multicast or broadcast capabilities.

Before a peer can use a rendezvous peer to propagate messages, it must connect to the rendezvous peer and obtain a lease. A lease specifies the amount of time that the peer requesting a connection to the rendezvous peer is allowed to use the rendezvous peer

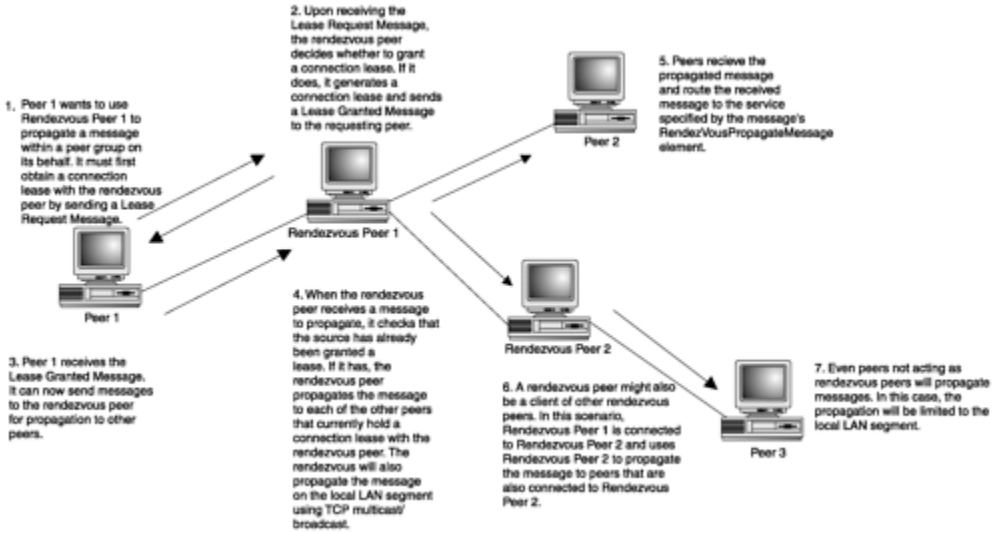
before it must renew the connection lease. To handle the interactions required to provide this functionality, the RVP defines three message formats:

- **Lease Request Message—** A message format used by a peer to request a connection lease to the rendezvous peer
- **Lease Granted Message—** A message format used by the rendezvous peer to approve a peer's Lease Request Message and provide the length of the lease
- **Lease Cancel Message—** A message format used by a peer to disconnect from the rendezvous peer

Unlike previous protocols, these messages are not specifically defined in terms of XML; instead, they are defined in terms of message elements. As in XML, message elements consist of a name and the contents of the element, and they can be nested. These message elements are used by the Endpoint service, discussed in Chapter 9, "The Endpoint Routing Protocol," to render messages into a format suitable for transmission over a specific network transport. Although the Endpoint service can render these message elements into XML, in most cases, it is more efficient to render the message elements into a more compact binary representation for transmission.

To connect with a rendezvous peer, a peer uses the sequence of messages shown in Figure 6.1.

Figure 6.1. Exchange of RVP messages.



Of course, before a peer can even begin the process of connecting to a rendezvous peer, it must discover the rendezvous peer by finding its Rendezvous Advertisement. After a rendezvous peer has been discovered, the peer sends requests to the rendezvous peer using the Endpoint service, addressing requests using `JxtaPropagate` as the service name and the ID of the peer group for which the peer is requesting rendezvous services as the service parameter. Endpoint service names and parameters are detailed in Chapter 9 's explanation of the Endpoint service.

The Rendezvous Advertisement

Peers that want to act as a rendezvous peer announce their capabilities to the network by publishing a Rendezvous Advertisement, as shown in Listing 6.1.

Listing 6.1 The Rendezvous Advertisement XML

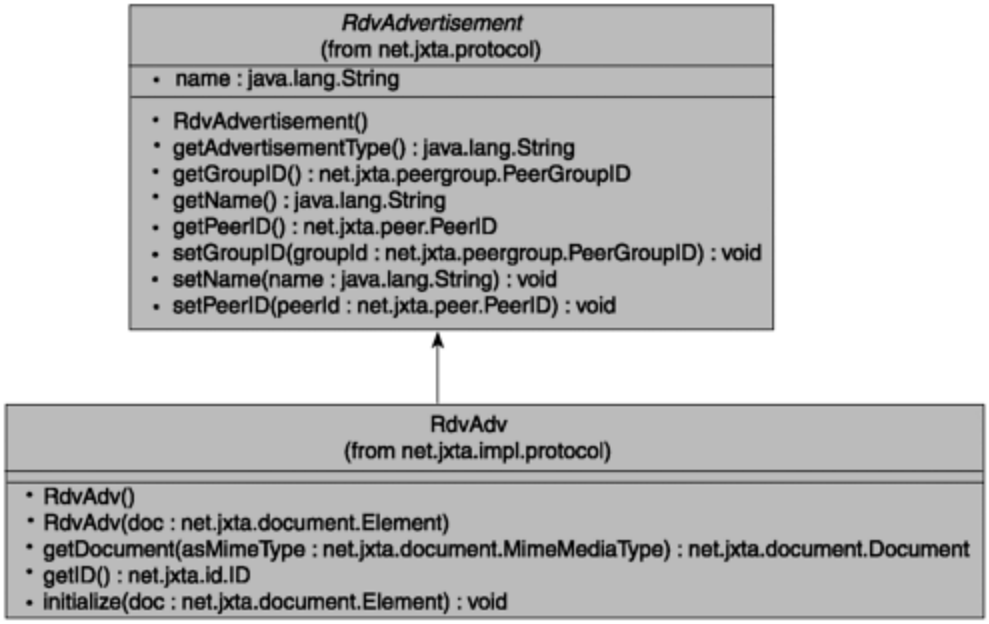
```
<?xml version="1.0"?>
<jxta:RdvAdvertisement xmlns:jxta="http://jxta.org">
  <RdvGroupId> . . . </RdvGroupId>
  <RdvPeerId> . . . </RdvPeerId>
  <Name> . . . </Name>
</jxta:RdvAdvertisement>
```

The Rendezvous Advertisement provides all the details that a peer needs to find a rendezvous peer to use to propagate messages on its behalf:

- **`RdvGroupId`**— A required element containing the ID of the peer group to which the peer is providing Rendezvous services.
- **`RdvPeerId`**— A required element containing the ID of the peer providing Rendezvous services to the specified peer group.
- **`Name`**— An optional element containing a symbolic name for the rendezvous peer. This name could be used by other peers to search for the rendezvous peer.

As shown in Figure 6.2, in the reference implementation, the Rendezvous Advertisement is represented by the `RdvAdvertisement` abstract class in the `net.jxta.protocol` package and is implemented by the `RdvAdv` class in `net.jxta.impl.protocol`. RdvAdvertisement

Figure 6.2. The Rendezvous Advertisement classes.



A peer can find rendezvous peers by sending a Discovery Query Message for Rendezvous Advertisements. To use the Discovery service in the reference implementation to search for Rendezvous Advertisements for a specific peer group, use this code:

```
discovery.getRemoteAdvertisements(null, 2, "RdvGroupId",
    currentGroup.getPeerGroupID().toString(),
    threshold, aListener);
```

This query searches for advertisements (type = `2`) that match the attribute `RdvGroupId` to the value of the given Peer Group ID. The responses to the Discovery Query Message are passed to the `DiscoveryListener` instance, `aListener`, for processing. The `DiscoveryService` instance used here is the Discovery service of the parent peer group used to create the peer group associated with the rendezvous peer.

Lease Request Message

When a peer has discovered a Rendezvous Advertisement and the rendezvous peer's corresponding Peer Advertisement, a peer can connect to the rendezvous peer and request a connection lease. If the rendezvous peer grants the request, the rendezvous peer adds the peer to its set of authorized peers. These authorized peers are allowed to use the rendezvous peer to propagate messages to other peers that are also connected to the rendezvous peer.

To request a connection lease from a rendezvous peer, a peer sends its own Peer Advertisement as the contents of a message element named `jxta:Connect`, as detailed in Table 6.1.

Table 6.1. The Lease Request Message

Element Name	Element Content
`jxta:Connect`	The Peer Advertisement of the peer requesting a connection lease from the rendezvous peer.

The Peer Advertisement content of the message element always is rendered as XML, independent of how the Endpoint service renders the message element. For example, the Endpoint service could render the Lease Grant Message as an XML message:

```
<jxta:Connect>
  <jxta:PA xmlns:jxta="http://jxta.org">
    . . .
  </jxta:PA>
</jxta:Connect>
```

The Endpoint service could even compress this string to produce a pure binary representation of the message element. However, regardless of how the message element is rendered, the message element's Peer Advertisement itself always is an XML document.

Lease Granted Message

If the rendezvous peer approves the peer's request for a connection lease, the requesting peer is added to the rendezvous peer's set of connected peers. The rendezvous peer responds to the requesting peer with a set of message elements collectively called the Lease Granted Message. The Lease Granted Message contains the rendezvous peer's Peer ID and a lease time, and it might contain the rendezvous peer's Peer Advertisement, as detailed in Table 6.2.

Table 6.2. The Lease Granted Message

Element Name	Element Content
`jxta:RdvAdvReply`	An optional message element containing the Peer Advertisement of the rendezvous peer granting the lease
`jxta:ConnectedPeer`	A required message element containing the Peer ID of the rendezvous peer granting the lease
`jxta:ConnectedLease`	A required message element containing a string representation of the lease time, in milliseconds

The lease time specifies the amount of time, in milliseconds, before a connected peer is removed from the rendezvous peer's set of connected peers. Peers that are connected to the rendezvous peer receive messages propagated by the rendezvous peer on behalf of other peers. Peers that are also located on the same LAN segment as the rendezvous peer receive the propagated message via TCP multicast. If a peer is located on the same LAN segment as the rendezvous peer, it receives a propagated message twice, once via direct communication by the rendezvous peer and once via TCP multicast.

Lease Cancel Message

When a peer no longer wants to use a rendezvous peer, it can cancel its connection lease, thereby removing itself from the rendezvous peer's set of connected peers. After it is removed, a peer can no longer use the rendezvous peer to propagate messages, nor will it receive messages propagated by the rendezvous peer on another peer's behalf.

To cancel the connection lease, a peer sends a message containing a `jxta:Disconnect` message element, as detailed in Table 6.3.

Table 6.3. The Lease Cancel Message

Element Name	Element Content
`jxta:Disconnect`	The Peer Advertisement of the peer requesting removal from the rendezvous peer's set of connected peers

The rendezvous peer removes the peer from its list of connected peers but does not provide any response to the peer.

Controlling Message Propagation

In Chapter 2, I noted the possibility for propagation to result in loopbacks, messages propagating infinitely between peer and rendezvous peers connected in a closed loop. As detailed in Chapter 2, loopback can be prevented by using a Time To Live (TTL) value that gets decremented each time a message is propagated. When the TTL reaches 0, the message is no longer propagated.

The RVP defines a message element to hold information that allows a rendezvous peer to detect loopback and discard the duplicate message. The contents of the message element include a RendezVous Propagate Message document, as shown in Listing 6.2.

Listing 6.2 The RendezVous Propagate Message XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:RendezVousPropagateMessage>
  <MessageId> . . . </MessageId>
  <DestSName> . . . </DestSName>
  <DestSParam> . . . </DestSParam>
  <TTL> . . . </TTL>
  <Path> . . . </Path>
</jxta:RendezVousPropagateMessage>
```

The contents of the RendezVous Propagate Message provide details about the service to which the message should be propagated and where the message has already been propagated:

- **`MessageId`**— A required element containing a unique identifier for the message being propagated. In the reference implementation, this is simply the time, in milliseconds, since the epoch, when the message is initially propagated. The reference implementation assumes the likelihood that two messages are being propagated within the same peer group at the same time and are sufficiently small to make this value unique.

- **`DestSName`**— A required element containing the name of the destination service for the propagated message.

- **`DestSParam`**— A required element containing the parameters for the destination service for the propagated message.

- **`TTL`**— A required element containing the propagated message's current TTL. The rendezvous peer discards the message if the message's TTL is `0`.

- **`Path`**— An optional element containing the Peer ID of a peer that the message being propagated has already visited. There can be more than one `Path` element, each specifying a waypoint in the message's propagation path. The rendezvous peer does not propagate a message to any peer that is contained in any of the RendezVous Propagate Message's `Path` elements.

For a message being propagated, the RendezVous Propagate Message is added to a message element with a name consisting of the concatenation of `RendezVousPropagate` and the ID of the peer group for which the rendezvous peer is providing Rendezvous services.

The Rendezvous Service

As shown in Figure 6.3, the Rendezvous service provides the implementation of the RVP, providing the functionality both to run a rendezvous peer and to propagate a message using a Rendezvous peer.

Figure 6.3. The Rendezvous service interfaces and classes.

When not configured to act as a rendezvous peer, a peer can use its Rendezvous service to propagate messages within a peer group using rendezvous peers to which it is connected. The peer can also use the Rendezvous service to propagate messages to peers in the same peer group using network transports that support multicasting within the LAN segment. When configured as a rendezvous peer, the Rendezvous service has the additional capability to propagate messages to other rendezvous and simple peers in the peer group on behalf of its set of connected peers.

Propagating Messages

The Rendezvous service's main functionality is to allow a peer to propagate messages to other peers on the network. This functionality is augmented when a Rendezvous service is configured to provide rendezvous peer services to other peers in the peer group. Regardless of whether a Rendezvous service is configured to provide rendezvous peer services, the `RendezVousService` interface provides three methods for propagating messages, as shown in Listing 6.3.

Listing 6.3 The *RendezVousService* Message Propagation Methods

```
public void propagate (Message msg, String serviceName,
    String serviceParam, int defaultTTL) throws IOException;
public void propagateInGroup (Message msg, String serviceName,
```

```
    String serviceParam, int defaultTTL, String prunePeer)
        throws IOException;
public void propagateToNeighbors (Message msg, String serviceName,
    String serviceParam, int defaultTTL, String prunePeer)
        throws IOException;
```

Each method provides a slightly different way of propagating a message to other peers in the peer group. All methods have the following parameters in common:

- **`msg`**— The `Message` object to be propagated to other peers.
- **`serviceName`**— A unique service name that identifies the service on the remote peer that is responsible for handling the `Message` object. In the reference implementation, this is set to the Module Class ID for the service. Modules and module classes are discussed in Chapter 10, "Peer Groups and Services."
- **`serviceParam`**— A parameter providing a name for a message queue. The service can use this parameter to route the handling of a message to the appropriate service instance.
- **`defaultTTL`**— A default Time To Live (TTL) value to be used when sending the `Message`. This TTL is used only if the `Message` doesn't currently have a TTL value. Otherwise, the propagation methods handle decrementing the `Message` 's TTL. In the reference implementation, the value passed as a default TTL can't be greater than the maximum TTL value of `10`. If the default TTL passed is larger than the maximum, it is set to the maximum TTL.

Both `propagateInGroup` and `propagateToNeighbors` take an extra argument, `prunePeer`, that specifies the Peer ID of a peer that should not be included in the propagation. The current reference implementation ignores this argument.

Each of the propagation methods has a slightly different purpose:

- **`propagateToNeighbors`**— This method propagates the `Message` to peers on the local network. In the reference implementation, a neighbor is a peer on the network that the rendezvous can communicate with directly, without going through

a router peer. This method relies on the Endpoint service to broadcast the message to peers on the local LAN segment using available network transports.

- **`propagateInGroup`**— This method propagates the `Message` to all peers in the peer group. This method duplicates the functionality of `propagateToNeighbors` but also propagates the given `Message` to each rendezvous peer with which the peer has a connection lease. If the local peer is configured to act as a rendezvous peer, this method also propagates the given `Message` to each peer that has a connection lease with the local peer.

- **`propagate`**— The documentation for the Java implementation gives the same description for this method as `propagateInGroup`. However, the reference implementation uses this method as a convenience method: When the passed `defaultTTL` is 1, `propagate` calls `propagateToNeighbors`. Otherwise, `propagate` calls `propagateInGroup`.

The propagation methods are all responsible not only for setting the `Message` 's TTL, but also for adding a message element containing the RendezVous Propagate Message to prevent against loopback.

Receiving Propagated Messages

The Rendezvous service is not responsible for blindly repropagating messages that it receives; instead, it serves only to propagate a message one network hop to another peer. It is the responsibility of a service on the peer to decide whether to repropagate the message. For example, the `ResolverService` implementation described in Chapter 5 repropagates a message only if the `QueryHandler` implementation's `processQuery` method doesn't throw a `DiscardQueryException`.

To allow a service to listen for propagated messages and decide whether to repropagate the message, the `RendezVousService` interface defines the `addPropagateListener` method:

```
public void addPropagateListener(String serviceNamePlusParameters,
    EndpointListener listener)
        throws IOException;
```

The `addPropagateListener` method registers an instance of `EndpointListener`, an interface described in Chapter 9, with the `RendezVousService` instance. The listener object is notified via its `processIncomingMessage` method when the Endpoint service receives a propagated message that matches the service name and parameters passed to `addPropagateListener`.

When an `EndpointListener` instance is notified of the arrival of a propagated message, it can repropagate the message by invoking the `propagateInGroup` method on a `RendezVousService` instance. The `RendezVousService` implementation handles updating the message's RendezVous Propagate Message with new TTL and path information.

When notification of propagated messages is no longer required, the `EndpointListener` instance can be unregistered from the `RendezVousService` using the `removePropagateListener` method:

```
public void removePropagateListener(String serviceNamePlusParameters,
    EndpointListener listener)
        throws IOException;
```

To remove a listener object successfully, the parameters passed to `removePropagateListener` must match those used to register the listener using `addPropagateListener`.

Connecting to and Disconnecting from Rendezvous Peers

The process of obtaining or cancelling a connection lease with a rendezvous peer is conducted entirely via the `RendezVousService` interface. To obtain a connection lease with a remote rendezvous peer, a peer invokes this code:

```
public void connectToRendezVous (PeerAdvertisement adv) throws
IOException;
```

Instead of using a Peer Advertisement, a peer can use an `EndpointAddress` to specify the remote rendezvous peer from which to obtain a connection lease:

```
public void connectToRendezVous (EndpointAddress addr) throws IOException;
```

The `EndpointAddress` is an abstraction of a network location that can be either network transport-neutral or transport-specific. Endpoint addresses are discussed in Chapter 9. When a peer has obtained a connection lease, the peer can cancel the lease granted by a remote rendezvous peer using this line:

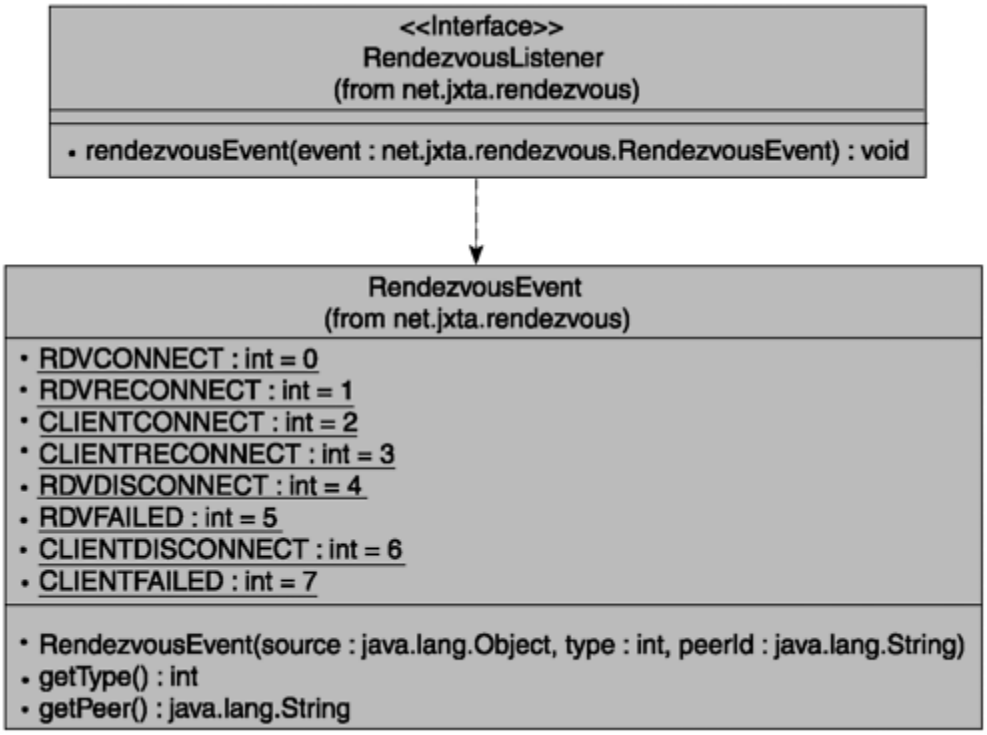
```
public void disconnectFromRendezVous (PeerID peerID);
```

Cancelling the lease with a rendezvous peer requires the Peer ID of the rendezvous peer.

The RendezvousListener and RendezvousEvent Classes

The Rendezvous service provides a `RendezvousListener` interface, as shown in Figure 6.4, that developers can implement to monitor the Rendezvous service. Registered `RendezvousListener` objects are notified when the Rendezvous service connects and disconnect from a rendezvous peer and when client peers connect or disconnect.

Figure 6.4. The `RendezvousListener` and `RendezvousEvent` classes.



The `RendezvousEvent` represents events fired by the `RendezVousService` when it is either acting as a client to another rendezvous peer or acting as a rendezvous peer to a client peer. The `RendezvousEvent.getType` method returns one of several possible values to inform the `RendezvousListener` what event has transpired. Type names starting with *CLIENT* indicate events triggered by the Rendezvous service handling a client peer message. Type names starting with *RDV* indicate events triggered by the Rendezvous service receiving a response to messages sent to a remote rendezvous peer. In total, eight possible values are returned by `RendezvousEvent.getType`:

- **`CLIENTCONNECT`**— The Rendezvous service has successfully processed a client peer's Connect request.

- **`CLIENTDISCONNECT`**— The Rendezvous service has successfully processed a client peer's Disconnect request.

- **`CLIENTRECONNECT`**— This is not currently used in the reference implementation. It indicates that the Rendezvous service has successfully processed a client peer's Connect request. In this case, the client peer was already

connected to the rendezvous peer but is connecting to renew its lease with the rendezvous peer. Most likely, this will be used when the lease time is used correctly.

- **`CLIENTFAILED`**— This also is not currently used in the reference implementation. It indicates that the Rendezvous service has unsuccessfully processed a client's Connect request.

- **`RDVCONNECT`**— The Rendezvous service, acting as a client peer, has received a response to its Connect request indicating that it is now connected to a rendezvous peer.

- **`RDVDISCONNECT`**— The Rendezvous service has successfully disconnected from a remote rendezvous peer. This event is not fired as a result of a response from the rendezvous peer, but it is fired immediately after the Disconnect request is sent to the rendezvous peer.

- **`RDVRECONNECT`**— This is not currently used in the reference implementation. It indicates that the Rendezvous service has received a response from a rendezvous peer confirming the success of a Connect request. In this case, the client peer was already connected to the rendezvous peer, but it sent a Connect to renew its lease with the rendezvous peer. Most likely, this will be used when the lease time is used correctly.

- **`RDVFAILED`**— This is not currently used in the reference implementation. The Rendezvous service has received a response indicating that its Connect request failed.

Implementations of `RendezvousListener` can be added to and removed from the `RendezVousService` instance using the `addListener` and `removeListener` methods. The methods operate in a similar fashion to `DiscoveryService` 's `addDiscoveryListener` and `removeDiscoveryListener` methods.

Support Classes Used by the Rendezvous Service

The `RendezVousService` relies on several other interfaces to abstract the task of managing peers' requests to obtain or cancel a connection lease. Each `RendezVousService` instance relies on an implementation of the `RendezVousManager` interface, as shown in Figure 6.5, to handle a client peer's request for a connection lease.

Figure 6.5. The `RendezVousManager` interface.

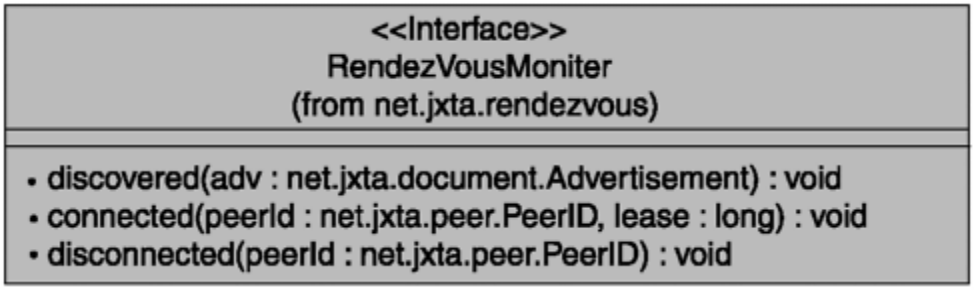
The `requestConnection` method processes the Peer Advertisement passed in the request and returns the lease time (in milliseconds). In the current reference implementation, the default lease time is 30 minutes, although this time is not currently used, as previously mentioned. A lease time of `0` indicates that the `RendezVousService` instance should not add the client to its set of connected client peers. A negative lease time indicates an infinite lease on the connection to the rendezvous peer.

Unlike the `RendezvousListener` interface, a `RendezVousService` instance has only one `RendezVousManager` instance. The `RendezVousManager` instance is initialized only when the `RendezVousService` is configured to act as a rendezvous for other peers. This `RendezVousManager` instance is passed to the `RendezVousService.startRendezVous` method used to start the Rendezvous service operating as rendezvous peer.

After the rendezvous peer is started using `startRendezVous`, the `RendezVousManager` instance can't be changed. Instead, the `RendezVousService` instance must be stopped using `stopRendezVous` and restarted using a different `RendezVousManager` instance. Stopping and starting the `RendezVousService` instance affects only the instance's operation as a rendezvous peer. The portion of `RendezVousService` instance responsible for allowing the local peer to propagate messages using other rendezvous peers is unaffected by `startRendezVous` and `stopRendezVous`.

Another support interface, `RdvMonitor`, provides functionality that is used when the Rendezvous service is acting as a client to a remote rendezvous peer. `RdvMonitor` 's methods, shown in Figure 6.6, are invoked when a client peer successfully obtains or cancels a connection lease with a rendezvous peer. A `RendezVousService` instance has only a single `RdvMonitor` instance.

Figure 6.6. The `RendezVousMonitor` interface.



When a peer receives a response indicating that a rendezvous peer has granted a connection lease, the `RdvMonitor.connected` method is invoked by the `RendezVousService` instance. The `connected` method accepts the rendezvous peer's Peer ID and the lease time for the connection. In the reference implementation, the `connected` method adds a local Rendezvous Advertisement for the remote peer and starts a thread to handle renewing the lease.

When a peer cancels a connection lease with a rendezvous peer, the `RendezVousService` instance invokes the `disconnected` method. Currently, the reference implementation of `RdvMonitor` doesn't do anything in the `disconnected` method.

The `RdvMonitor` interface defines one other method, `discovered`, which is invoked by the `RendezVousService` instance to provide an advertisement for other rendezvous peers. When the `RendezVousService` 's `sendRendezVousAdv` method is called, the peer sends a message containing a message element named `jxta:RdvAdv` that contains a Peer Advertisement. This Peer Advertisement is for a rendezvous peer that the `RendezVousService` instance wants to publish to other peers. When a peer's `RendezvousService` receives a message containing a `jxta:RdvAdv` message element, the service's `RdvMonitor` instance has its `discovered` method invoked. This feature can be

used to distribute the load away from a particular rendezvous peer. Currently, the reference implementation simply publishes the advertisement locally when `discovered` is called.

Unlike `RendezVousManager`, the `RdvMonitor` instance can be set using the `RendezVousService.setMonitor` method.

Other Useful RendezVousService Methods

The `RendezVousService` interface defines several other useful methods:

- **`getConnectedPeers`**— Returns an `Enumeration` of `ID` s of all the client peers currently connected to the rendezvous peer. When a peer is not acting as a rendezvous peer, the `Enumeration` is empty.
- **`getConnectedRendezVous`**— Returns an `Enumeration` of `ID` s of all the rendezvous peers to which the peer is connected. This method returns results regardless of whether the peer is operating as a rendezvous peer.
- **`getDisconnectedRendezVous`**— Returns an `Enumeration` of `ID` s of all rendezvous peers to which the peer has failed to connect.
- **`isConnectedToRendezVous`**— Returns `true` if the peer is currently connected to at least one rendezvous peer.
- **`isRendezVous`**— Returns `true` if the peer is providing rendezvous peer services to other client peers in Rendezvous service's peer group.

Maintaining Rendezvous Connections

To ensure that a peer receives propagated messages, the peer must maintain its connection lease to a number of rendezvous peers. Without maintaining and renewing the connection lease, a peer risks not receiving propagated messages from members of its peer group that are not part of its local LAN segment.

To address this issue, the reference implementation provides the `RendAddrCompactor` class in `net.jxta.impl.rendezvous`. The `RendAddrCompactor` class runs a thread that

regularly uses the Discovery service to find Rendezvous Advertisements and maintain connections to rendezvous peers. The `RendAddrCompactor` thread tries to discover and obtain a connection lease with up to three rendezvous peers, thereby attempting to guarantee connectivity with peer group members outside the local LAN segment.

Summary

In this chapter, you saw how the Rendezvous service allows peers to propagate messages to other peers within a peer group.You also learned that the Rendezvous service provides the capability for a peer to act as a rendezvous peer and propagate messages on behalf of other peers in a peer group. Developers can use the Rendezvous service not only to send messages, but also to provide their own custom functionality when client peers connect to the Rendezvous service to obtain rendezvous peer services.

In the next chapter, you explore the Peer Information Protocol, which is a protocol that monitors peers and obtains peer status information. The Peer Information Protocol allows a peer to gather information about a remote peer that it can use to determine the suitability of the peer for performing a task.

Chapter 7. The Peer Information Protocol

The peer information protocol (PIP) allows peers to obtain status information from previously discovered peers. This status information is currently limited to include only data on the uptime of the peers and the amount of traffic processed by the peer. Future work on the PIP will most likely extend this basic protocol to provide ways for developers to extend the protocol's default status-monitoring capabilities.

Introducing the Peer Information Protocol

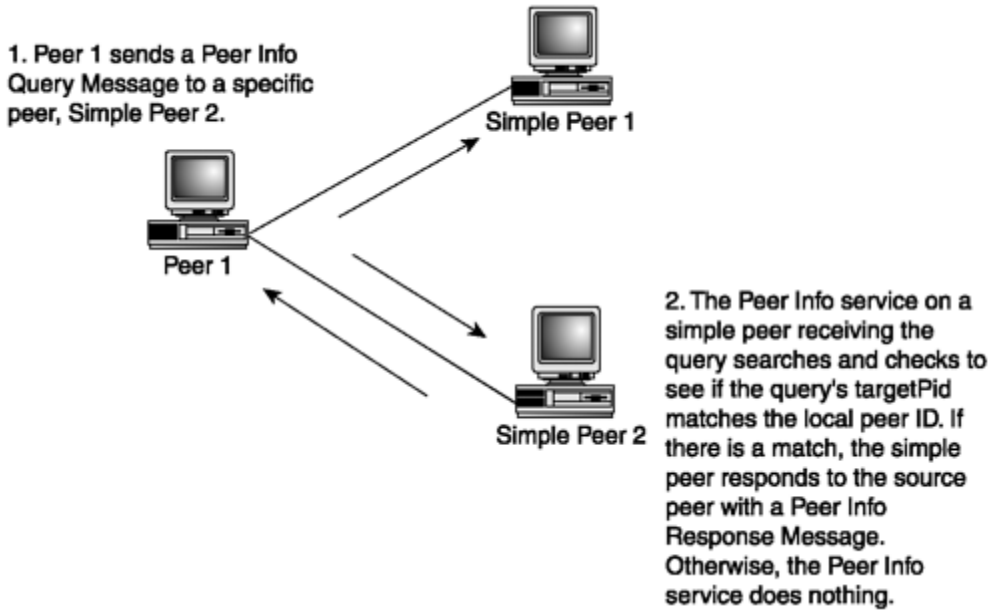
After a remote peer has been discovered using the Discovery service and the Peer Discovery Protocol, a peer might want to monitor the remote peer's status to make additional decisions about how to use the remote peer most effectively or to make the use of its services by other peers more efficient. Monitoring is an essential part of a P2P network, providing information that peers can use to leverage the resources of the P2P network in the most efficient manner. For example, in a file-sharing P2P application, a peer could use status information describing the current network traffic on a remote peer to decide whether to use the remote peer as a source of services. If the remote peer is under an extreme load, it's in the interests of both the client peer and the P2P network in general to shift usage away from that remote peer.

The Peer Information Protocol (PIP) is an optional JXTA protocol that allows a peer to monitor a remote peer and obtain information on the remote peer's current status. As with all the protocols described up to this point in the book, the PIP requires only two types of messages:

- **Peer Info Response Message—** A message format for providing a peer's status
- **Peer Info Query Message—** A message format for querying a remote peer's status to other peers

These two messages are responsible for providing access to a peer's status information using the protocol shown in Figure 7.1.

Figure 7.1. Exchange of Peer Info messages.



Although Figure 7.1 shows that a peer that receives a Peer Info Query Message not addressed to it does nothing with the message, the reference implementation is slightly different. The reference implementation provides for the possibility that a query message might be propagated to a peer instead of sent to a specific peer. If a peer receives a query to which it isn't the subject of the query, the peer propagates the query to the peer group.

The Peer Info service implements the PIP by leveraging the Resolver and Rendezvous services. This implementation follows a similar pattern to the Discovery service. The Peer Info service uses Resolver Query and Response Messages and the Resolver service to handle the details of sending a query to a named handler and generating a response. The Resolver service is responsible for handling the details of propagating messages to other simple peers and rendezvous peers for the Resolver service. As with all the protocols in JXTA that you've seen so far, a PIP query to a remote peer might not result in a response.

The Peer Info Query Message

The Peer Info Query Message is very simple, if not a little limited, as shown in Listing 7.1.

Listing 7.1 The Peer Info Query Message

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PeerInfoQueryMessage xmlns:jxta="http://jxta.org">
  <sourcePid> . . . </sourcePid>
  <targetPid> . . . </targetPid>
  <request> . . . </request>
</jxta:PeerInfoQueryMessage>
```

The Peer Info Query Message specifies three parameters:

- **`sourcePid`**— A required element containing the ID of the peer requesting status information from a remote peer. This Peer ID is a string encoding of the JXTA URN for the peer that generated the query.

- **`targetPid`**— A required element containing the ID of the remote peer from which status information is being solicited. This Peer ID is a string encoding of the JXTA URN for the peer that is the target of the query.

- **`request`**— An optional element containing a string specifying the status information being requested from the remote peer. The format of this request string is unspecified; it is the responsibility of the recipient to know how to decode it.

Unfortunately, the current reference implementation provides no mechanism to allow a developer to handle requests specified by the contents of the `request` element in the query. It appears that this is work that will be undertaken in the future to allow developers to add their own code to unmarshall the contents of the `request` element and provide response information to the peer requesting status information.

Unlike other protocols that you've seen so far, each rendezvous peer that propagates this message does not generate a response to the query. When the Peer Info service receives a Peer Info Query Message, it checks to see if the local peer's ID matches the `targetPid`. If the IDs match, the service generates a response that is sent by the Resolver service to the source peer. Otherwise, no response is generated and the message is propagated to other peers that might be capable of providing the response. Theoretically, it should be possible to propagate a Peer Info Query Message in the reference implementation, but functionality to handle this case has been added as a precaution.

The Peer Info Query Message is different from the other protocol implementations in another significant way: The abstract `PeerInfoQueryMessage` class in net.jxta.protocol and the reference implementation `PeerInfoQueryMsg` in net.jxta.impl.protocol aren't used throughout by the reference implementation! Although some areas of the reference implementation do use these objects, their use is inconsistent.

This inconsistency suggests that the implementation of the PIP is in a much earlier stage of development compared to some of the other protocols. The lack of a mechanism to allow a developer to handle the contents of the `request` element of the Peer Info Query Message further underlines the fact that the PIP is still a work in progress.

The Peer Info Response Message

The counterpart to the Peer Info Query Message, the Peer Info Response Message, is significantly more detailed, as shown in Listing 7.2.

Listing 7.2 The Peer Info Response Message

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PeerInfoResponse xmlns:jxta="http://jxta.org">
  <sourcePid> . . . </sourcePid>
  <targetPid> . . . </targetPid>
  <uptime> . . . </uptime>
  <timestamp> . . . </timestamp>
  <response> . . . </response>
  <traffic>
    . . .
  </traffic>
</jxta:PeerInfoResponse>
```

The Peer Info Response Message provides a variety of status information, most of which is oriented to providing information on the network traffic load on the remote peer:

- **`sourcePid`**— A required element containing the ID of the peer requesting status information from the peer. This Peer ID is a string encoding of the JXTA URN for the peer that generated the original Peer Info Query Message.

- **`targetPid`**— A required element containing the ID of the remote peer from which status information is being solicited. This Peer ID is a string encoding of the JXTA URN for the peer providing the response to the Peer Info Query Message.

- **`uptime`**— An optional element containing the amount of time, in milliseconds, since the peer joined the P2P network. In the reference implementation, the uptime corresponds to the amount of time that has elapsed since the Peer Info service started.

- **`timestamp`**— An optional element containing a timestamp describing the time when the peer generated the status information contained in the response. This timestamp is given in milliseconds since the epoch (January 1, 1970, 00:00:00 GMT).

- **`response`**— An optional element containing a string specifying the status information being returned in response to the query's `request` element's content. The format of this response string is unspecified; it is the responsibility of the recipient to know how to decode it. The reference PIP implementation does not currently provide a mechanism to allow a developer to populate the `response` element to provide the requested information.

- **`traffic`**— An optional element that contains details on the network traffic handled by the peer. The format of the contents of the `traffic` element is shown in Listing 7.3.

Listing 7.3 The Format of the *traffic* Element Contents

```
<traffic>
  <lastIncomingMessageAt> . . . </lastIncomingMessageAt>
  <lastOutgoingMessageAt> . . . </lastOutgoingMessageAt>
  <in>
    <transport endptaddr=" . . . "> . . . </transport>
  </in>
  <out>
    <transport endptaddr=" . . . "> . . . </transport>
  </out>
</traffic>
```

The contents of the `traffic` element in the Peer Info Response Message describe in detail the network traffic handled by the peer:

- **`lastIncomingMessageAt`**— An optional element containing a timestamp specifying the last time that the peer's endpoints handled an incoming message. The timestamp is given in milliseconds since the epoch.

- **`lastOutgoingMessageAt`**— An optional element containing a timestamp specifying the last time that the peer's endpoints handled an outgoing message. The timestamp is given in milliseconds since the epoch.

- **`in`**— An optional element containing details on the inbound traffic seen by the peer's endpoints. The `in` element may contain zero or more `transport` elements.

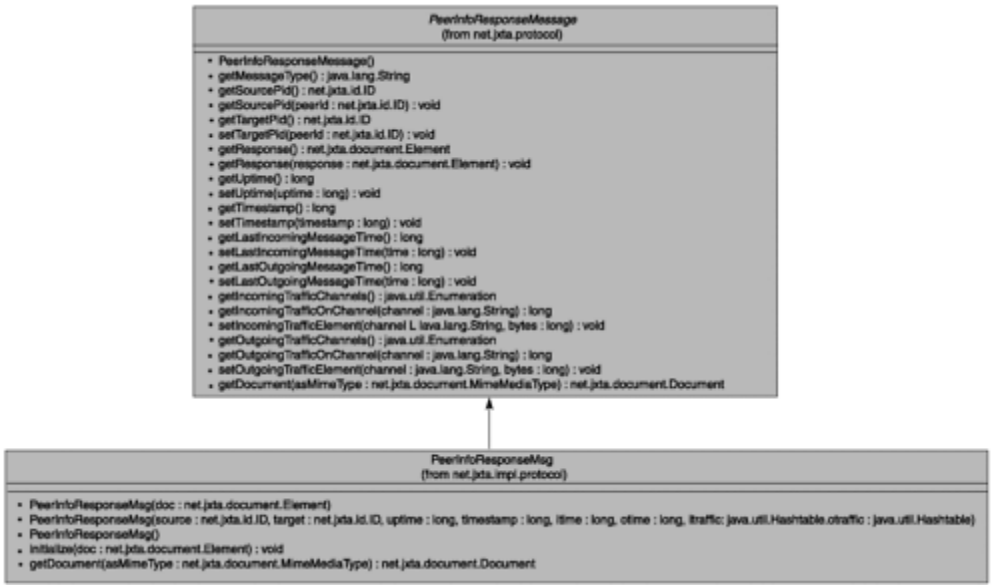
- **`transport`**— An optional element containing the number of bytes processed by the endpoint address specified by the `endptaddr` attribute. When used inside the `in` element, this element specifies the number of bytes received by the endpoint address specified. When used inside the `out` element, this element specifies the number of bytes sent by the endpoint address specified. The format of the endpoint address is covered in Chapter 9, "The Endpoint Routing Protocol."

- **`out`** — A container element to hold details on the outbound traffic seen by the peer's endpoints. The `out` element may contain zero or more `transport` elements.

The reference implementation of the PIP has one oversight in its current form: Peer Info Query Messages are propagated indiscriminately. When a peer receives a Peer Info Query in which the `targetPid` matches its local Peer ID, it generates a Peer Info Response Message that the Resolver service sends to the peer that generated the query. Unfortunately, the Resolver service still propagates the query, even though the target peer has responded.

Similar to the Peer Info Query Message, the reference implementation provides the `PeerInfoResponseMessage` abstract class in net.jxta.protocol and the `PeerInfoResponseMsg` implementation class in net.jxta.impl.protocol. These classes are shown in Figure 7.2.

Figure 7.2. The Peer Info Response Message classes.

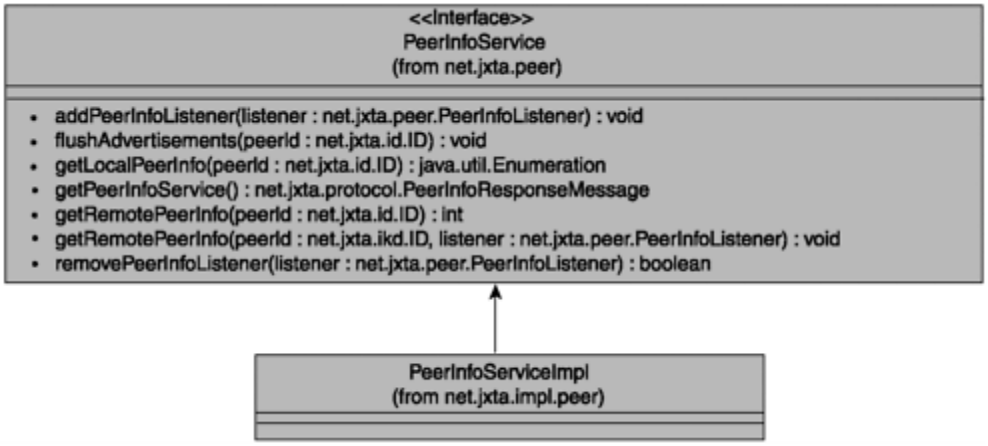


Unlike the `PeerInfoQueryMessage` and `PeerInfoQueryMessage` classes, the `PeerInfoResponseMessage` and `PeerInfoResponseMsg` classes are used throughout the reference implementation to handle parsing and formatting Peer Info Response Messages.

The Peer Info Service

As with the other protocols, the PIP is encapsulated as a service, as shown in Figure 7.3, freeing the developer from dealing with the details of the Peer Info Query and Response Messages. PeerInfoResponseMessage

Figure 7.3. The Peer Info service interface and implementation.



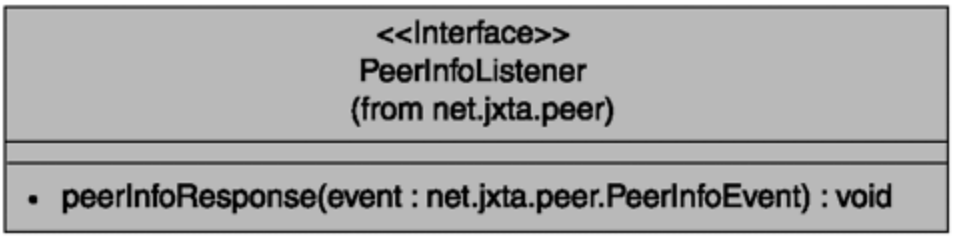
The definition of the `PeerInfoService` interface is very similar to the `DiscoveryService` interface, providing methods to retrieve remote and local peer status information. Like the Discovery service, the Peer Info service provides a mechanism to register a listener that will be notified when the Peer Info service receives a Peer Info Response Message.

The reference implementation of the Peer Info service is a `QueryHandler` implementation, whose `processQuery` is responsible for generating a response, if any. Unfortunately, there is no way for the `processQuery` implementation to signal to the Resolver service that a response has been generated and that the original query should not be propagated. If the `processQuery` implementation threw a `DiscardResponseException`, the Resolver service wouldn't propagate the query. However, throwing this exception would prevent `processQuery` from returning a response to be sent to the source of the original query. This incapability to prevent propagation is responsible for the current reference implementation's undesirable propagation of Peer Info Query Messages.

The PeerInfoListener Interface

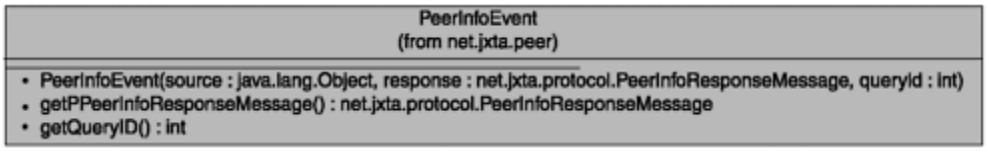
As shown in Figure 7.4, to receive notifications of incoming Peer Info Response Messages, a developer can create and register an implementation of the `PeerInfoListener` interface.

Figure 7.4. The `PeerInfoListener` interface.



Like `DiscoveryListener`, `PeerInfoListener` provides only one method that gets invoked when the Peer Info service receives a Peer Info Response Message. The `peerInfoResponse` method accepts a `PeerInfoEvent` object, shown in Figure 7.5, that can be used by a `PeerInfoListener` implementation to extract the Peer Info Response Message.

Figure 7.5. The `PeerInfoEvent` class.



The `PeerInfoEvent` 's `getPPeerInfoResponseMessage` returns a `PeerInfoResponseMessage` instance that contains the response received from a remote peer to a Peer Info Query Message.

Using the Peer Info Service

To demonstrate the use of the Peer Info service, you'll implement a simple `PeerInfoListener` and use the peer group's `PeerInfoService` instance to obtain a remote peer's status information.

By this point in the book, you've probably noticed a number of common architectural patterns employed by the JXTA reference implementation. These patterns include the use of listener objects to provide callback functionality, the division of all implementations into an abstract class defining the Java implementation's API, and a concrete class providing the reference implementation. Because of the similarities between the Peer Info

and Discovery services, this example skims over some of the basic details that were explained in Chapter 4, "The Peer Discovery Protocol."

Implementing PeerInfoListener

Implementing the `PeerInfoListener` interface is very similar to implementing the `DiscoveryListener` interface. A developer needs only to create a class that implements the `peerInfoResponse` method, as shown in Listing 7.4, and register an instance of the implementation with the Peer Info service.

Listing 7.4 Source Code for *ExampleListener.java*

```
package net.jxta.impl.shell.bin.example7_1;

import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.StructuredTextDocument;
import net.jxta.document.MimeMediaType;

import net.jxta.peer.PeerInfoEvent;
import net.jxta.peer.PeerInfoListener;

import net.jxta.protocol.PeerInfoResponseMessage;



/**
 * A simple implementation of PeerInfoListener to print out details on
 * the received Peer Info Response Messages.
 */
public class ExampleListener implements PeerInfoListener
{
    /**
     * A simple handler that prints out the details on the received
```

```
     * peer status information.
     *
     * @param   event the event detailing the received response.
     */
    public void peerInfoResponse(PeerInfoEvent event)
    {
        // Extract the peer info response from the event.
        PeerInfoResponseMessage response =
            event.getPPeerInfoResponseMessage();

        // Print out the peer info.
        System.out.println("Uptime: " + response.getUptime());
        System.out.println("Timestamp: " + response.getTimestamp());
        System.out.println("Target: " + response.getTargetPid());
        System.out.println("Source: " + response.getSourcePid());
        System.out.println("Last Incoming Message: "
            + response.getLastIncomingMessageTime());
        System.out.println("Last Outcoming Message: "
            + response.getLastOutgoingMessageTime());

        // Print out the incoming channel statistics.
        Enumeration incoming = response.getIncomingTrafficChannels();
        if (incoming != null)
        {
            while (incoming.hasMoreElements())
            {
                String incomingchannel = (String) incoming.nextElement();
                long incomingbytes =
                    response.getIncomingTrafficOnChannel(incomingchanne
l);

                System.out.println(
                    incomingbytes + " incoming bytes on channel "
                        + incomingchannel);
            }
        }
```

```
        // Print out the outgoing channel statistics.
        Enumeration outgoing = response.getOutgoingTrafficChannels();
        if (outgoing != null)
        {
            while (outgoing.hasMoreElements())
            {
                String outgoingchannel = (String) outgoing.nextElement();
                long outgoingbytes =
                    response.getOutgoingTrafficOnChannel(outgoingchanne
l);

                System.out.println(
                    outgoingbytes + " incoming bytes on channel "
                        + outgoingchannel);
            }
        }

        System.out.println("Done with status info...");
    }
}
```

Extracting the Peer Info Response Message is as simple as a call to

```
getPPeerInfoResponseMessage:
    PeerInfoResponseMessage peerinfo =
        event.getPPeerInfoResponseMessage();
```

The returned `PeerInfoResponseMessage` can then be used to obtain the `timestamp`, `uptime`, and other Peer Info Response Message elements' contents.

Registering a PeerInfoListener

Before a `PeerInfoListener` implementation will begin receiving notification of incoming Peer Info Response Messages, the implementation must be registered with the Peer Info

service for a specific peer group. A `PeerInfoListener` is registered using the `PeerInfoService` interface `addPeerInfoListener` method:

```
public void addPeerInfoListener (PeerInfoListener listener)
```

Like the Discovery service, the reference implementation of the Peer Info service is implemented as a Resolver handler. The `PeerInfoServiceImpl` implements the `QueryHandler` interface and handles invoking the registered listeners' `peerInfoResponse` method.

An alternative to registering a handler with the `PeerInfoService` is to pass a `PeerInfoListener` instance when querying remote peers for status information. The `PeerInfoService.getRemotePeerInfo` method accepts a `PeerInfoListener` instance:

```
peerinfo.getRemotePeerInfo(peerIdObject, new ExampleListener());
```

If `getRemotePeerInfo` is called with a `PeerInfoListener` implementation, the given listener object is invoked when replies for the query arrive. In the reference implementation of `PeerInfoService`, `PeerInfoServiceImpl`, the listener is stored to a `Hashtable` using a query ID as the key. The query ID is used in the creation of the Resolver Query Message, and the response sent by a remote peer should use the same query ID. When a Peer Info Response message arrives, wrapped in a Resolver Response Message, the `PeerInfoServiceImpl` extracts the query ID from the `ResolverResponseMsg` and uses it to find a listener in the `Hashtable` with the matching query ID. If a listener is found, its `peerInfoResponse` method is invoked. This is done in addition to invoking the `peerInfoResponse` method of all the listeners registered using `addPeerInfoListener`.

As with the Discovery service, listeners can be removed from the `PeerInfoService` instance. Removing a listener stops it from receiving notification of new incoming Peer Info Response Messages. To remove a listener, a reference to the listener object is required:

```
public boolean removePeerInfoListener ( PeerInfoListener listener);
```

The `removePeerInfoListener` method returns `true` if the `PeerInfoService` has successfully removed the listener. If the method returns `false`, it indicates that the listener object could not be found in the service's set of registered listeners. Unfortunately, listeners that are added to the service by invoking `getRemotePeerInfo` with a listener object cannot be removed using `removePeerInfoListener`.

Finding Remote Peer Information

Using the `ExampleListener` shown in Listing 7.4, it's simple to create a Shell command to send a query to remote peers for peer status information, shown in Listing 7.5.

Listing 7.5 Source Code for *example7_1.java*

```
package net.jxta.impl.shell.bin.example7_1;

import java.net.URL;

import net.jxta.id.IDFactory;
import net.jxta.peer.PeerID;
import net.jxta.peer.PeerInfoService;

import net.jxta.peergroup.PeerGroup;

import net.jxta.impl.shell.GetOpt;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;


/**

 * A simple Shell command to demonstrate the use of the Peer Info
 * service and the PeerInfoListener interface to query remote peers for
 * status info.
 */
```

```
public class example7_1 extends ShellApp
{
    /**
     * The ID of the peer from whom peer info is being solicited.
     */
    private String peerid = null;


    /**
     * Parses the command-line arguments and initializes the command
     *
     * @param       args the arguments to be parsed.
     * @exception   IllegalArgumentException if an invalid parameter
     *              is passed.
     */
    private void parseArguments(String[] args)
        throws IllegalArgumentException
    {
        int option;

        // Parse the arguments to the command.
        GetOpt parser = new GetOpt(args, "p:");

        while ((option = parser.getNextOption()) != -1)
        {
            switch (option)
            {
                case 'p' :
                {
                    // Set the ID of the peer used to retrieve peer info.
                    peerid = parser.getOptionArg();

                    break;
                }
            }
        }
```

```
    }

    /**
     * Sends a query to a remote peer.
     *
     * @param   aPeerId the ID of the peer from whom to solicit status info.
     * @param   peerinfo the PeerInfoService to use to perform the query,
     */
    private void sendRemoteRequest(String aPeerId, PeerInfoService
peerinfo)
    {
        try
        {
            // Transform the Peer ID string into a Peer ID object.
            PeerID peerIdObject = (PeerID) IDFactory.fromURL(
                new URL((aPeerId)));

            // Use the Peer Info service to query for the peer info.
            peerinfo.getRemotePeerInfo(peerIdObject, new
ExampleListener());
        }
        catch (Exception e)
        {
            System.out.println("Error parsing Peer ID string: " + e);
        }
    }

    /**
     * The implementation of the Shell command, invoked when the command
     * is started by the user from the Shell.
     *
     * @param   args the command-line arguments passed to the command.
     * @return  a status code indicating the success or failure of
     *          the command.
     */
    public int startApp(String[] args)
```

```
    {
        String peerid = null;
        int result = appNoError;

        // Get the shell's environment.
        ShellEnv theEnvironment = getEnv();

        // Use the environment to obtain the current peer group.
        ShellObject theShellObject = theEnvironment.get("stdgroup");
        PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

        // Get the Peer Info service for the current peer group.
        PeerInfoService peerinfo = currentGroup.getPeerInfoService();

        // Default to getting the local peer's status info.
        peerid = currentGroup.getPeerID().toString();

        try
        {
            // Parse the command-line arguments.
            parseArguments(args);
        }
        catch (IllegalArgumentException e)
        {
            println("Incorrect parameters passed to the command.");
            result = ShellApp.appParamError;
        }

        // Send a remote peer info request.
        System.out.println("Running example7_1...");
        sendRemoteRequest(peerid, peerinfo);

        return result;
    }
}
```

The `example7_1` command sends a query to a remote peer using the `PeerInfoService` 's `getRemotePeerInfo` method, passing an instance of the `ExampleListener` class to handle the responses.

It's important to note that `getRemotePeerInfo` requires a real Peer ID to be passed to the method. Although the reference implementation allows `null` to be passed to `getRemotePeer`, the `targetPid` element for the Peer Info Query Message is empty. The reference implementation of Peer Info service that receives the query checks the `targetPid` against the local Peer ID and, because they don't match, does nothing. As a result, a response is never generated in response to the query.

By default, `example7_1` uses the local Peer ID as the `targetPid`, resulting in the status information for the local peer. To retrieve remote peer information, the command must be invoked with a Peer ID, such as in this code:

```
JXTA>example7_1
-purn:jxta:uuid-59616261646162614A787461503250332A2C6697AF84
4127A89E8F30B01CA1C403
```

To use `example7_1` to retrieve the peer information for a specific peer, you first need the ID of the remote peer. Run the `peers` command and choose a peer from which you want to solicit status information.View the peer's advertisement using `cat`:

```
JXTA>cat peer0
```

Find the `PID` element in the Peer Advertisement, and use that value to invoke the `example7_1` command with the `-p` option. Unfortunately, the Shell doesn't currently support paste operations on all platforms, so you can't cut and paste the `PID` value.

An easier way to invoke the command is by using a *Shell script*. A Shell script is any plain text file that contains Shell commands. To run a Shell script, do the following:

```
JXTA>Shell -ftest.txt
```

This example runs the script `test.txt` from the Shell's current directory. In the Shell, the current directory is the Shell subdirectory of the JXTA Demo install directory. To run the `example7_1` command from a script, follow these steps:

1. Create a text file in the Shell subdirectory of the JXTA Demo installation directory. For this example, call the file `test.txt`.

2. Use the right-click pop-up menu in the Shell to copy the `PID` from the remote peer that you want to query.

3. Edit `test.txt`.

4. Add the text **example7_1 –p** to the text file, and then paste the `PID` directly after the `-p` option.

5. Save the `test.txt` file.

6. From the Shell, run the script using this command:

```
JXTA>Shell -ftest.txt
```

This runs the `example7_1` command and queries the peer specified by the `PID` that you entered in the script. When the `ExampleListener` receives responses, the peer information details are printed to the console (not the Shell console) and resemble Listing 7.6.

Listing 7.6 Example Output from *ExampleListener*

```
Uptime: 7330
Timestamp: 1007439754658
Target: urn:jxta:uuid-59616261646162614A787461503250332A2C6697AF84
4127A89E8F30B01CA1C403
Source: urn:jxta:uuid-59616261646162614A78746150325033AEB5D26090CD4EC683
E18ABE877ABE2703
Last Incoming Message: 0
Last Outcoming Message: 0
Done with status info...
```

Note

As of build 49b of the Java reference implementation of the JXTA platform, the `PeerInfoService` implementation is disabled. This is a result of ongoing work to resolve issues within the implementation. If the examples in this chapter do not produce any output, it is most likely due to this ongoing development work. Consult the `platform.jxta.org` web site for more information on the current status of the PIP implementation.

Finding Cached Peer Information

Just as the Discovery service enables you to query the local cache of advertisements, the Peer Info service provides a mechanism for retrieving cached status information. The example shown in Listing 7.7 provides a simple command for retrieving the locally cached peer information using the Peer Info service.

Listing 7.7 Source Code for *example7_2.java*

```
package net.jxta.impl.shell.bin.example7_2;

import java.io.IOException;
import java.io.StringWriter;

import java.net.URL;

import java.util.Enumeration;

import net.jxta.document.Advertisement;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.MimeMediaType;

import net.jxta.id.IDFactory;

import net.jxta.peer.PeerID;
import net.jxta.peer.PeerInfoListener;
```

```
import net.jxta.peer.PeerInfoService;

import net.jxta.peergroup.PeerGroup;

import net.jxta.protocol.PeerInfoResponseMessage;

import net.jxta.impl.shell.GetOpt;
import net.jxta.impl.shell.ShellApp;
import net.jxta.impl.shell.ShellEnv;
import net.jxta.impl.shell.ShellObject;


/**
 * A simple Shell command to demonstrate the use of the Peer Info
 * service and the PeerInfoListener interface to retrieve locally cached
 * status info.
 */
public class example7_2 extends ShellApp
{
    /**
     * The ID of the peer from which peer info is being solicited.
     */
    private String peerid = null;
    /**
     * Parses the command-line arguments and initializes the command
     *
     * @param       args the arguments to be parsed.
     * @exception  IllegalArgumentException if an invalid parameter
     *             is passed.
     */
    private void parseArguments(String[] args)
        throws IllegalArgumentException
    {
        int option;

        // Parse the arguments to the command.
```

```
        GetOpt parser = new GetOpt(args, "p:l");

        while ((option = parser.getNextOption()) != -1)
        {
            switch (option)
            {
                case 'p' :
                {
                    // Set the ID of the peer used to retrieve peer info.
                    peerid = parser.getOptionArg();

                    break;
                }
            }
        }
    }

    /**
     * Retrieves peer information from the local cache.
     *
     * @param   aPeerId the ID of the peer from which to solicit status info.
     * @param   peerinfo the PeerInfoService to use to perform the query,
     */
    private void sendLocalRequest(String aPeerId, PeerInfoService
peerinfo)
    {
        try
        {
            PeerID peerIdObject = (PeerID) IDFactory.fromURL(
                new URL((aPeerId)));
            Enumeration enum = peerinfo.getLocalPeerInfo(peerIdObject);

            // Iterate through the response messages.
            while (enum.hasMoreElements())
            {
                // Extract the peer info response from the event.
```

```
                PeerInfoResponseMessage response =
                    (PeerInfoResponseMessage) enum.nextElement();

                // Print out the peer info.
                System.out.println("Uptime: " + response.getUptime());
                System.out.println("Timestamp: " +
response.getTimestamp());
                System.out.println("Target: " + response.getTargetPid());
                System.out.println("Source: " + response.getSourcePid());
                System.out.println("Last Incoming Message: "
                    + response.getLastIncomingMessageTime());
                System.out.println("Last Outcoming Message: "
                    + response.getLastOutgoingMessageTime());

                // Print out the incoming channel statistics.
                Enumeration incoming =
                    response.getIncomingTrafficChannels();
                if (incoming != null)
                {
                    while (incoming.hasMoreElements())
                    {
                        String incomingchannel =
                            (String) incoming.nextElement();
                        long incomingbytes =
                            response.getIncomingTrafficOnChannel(
                                incomingchannel);

                        System.out.println(incomingbytes
                            + " incoming bytes on channel "
                            + incomingchannel);
                    }
                }

                // Print out the outgoing channel statistics.
                Enumeration outgoing =
                    response.getOutgoingTrafficChannels();
```

```
            if (outgoing != null)
            {
                while (outgoing.hasMoreElements())
                {
                    String outgoingchannel =
                        (String) outgoing.nextElement();
                    long outgoingbytes =
                        response.getOutgoingTrafficOnChannel(
                            outgoingchannel);

                    System.out.println(outgoingbytes
                        + " incoming bytes on channel "
                        + outgoingchannel);
                }
            }

            System.out.println("Done with status info...");
        }
    }
    catch (IOException e)
    {
        println("Error retrieving local peer info responses!" + e);
    }
}

/**
 * The implementation of the Shell command, invoked when the command
 * is started by the user from the Shell.
 *
 * @param  args the command-line arguments passed to the command.
 * @return  a status code indicating the success or failure of
 *         the command.
 */
public int startApp(String[] args)
{
    String peerid = null;
```

```
        int result = appNoError;

        // Get the shell's environment.
        ShellEnv theEnvironment = getEnv();

        // Use the environment to obtain the current peer group. continues
        ShellObject theShellObject = theEnvironment.get("stdgroup");
        PeerGroup currentGroup = (PeerGroup) theShellObject.getObject();

        // Get the PeerInfo service for the current peer group.
        PeerInfoService peerinfo = currentGroup.getPeerInfoService();

        // Default to getting the local peer's status info.
        peerid = currentGroup.getPeerID().toString();

        try
        {
            // Parse the command-line arguments.
            parseArguments(args);
        }
        catch (IllegalArgumentException e)
        {
            println("Incorrect parameters passed to the command.");
            result = ShellApp.appParamError;
        }

        peerid = null;

        // Send a local peer info request.
        System.out.println("Running example7_2...");
        sendLocalRequest(peerid, peerinfo);

        return result;
    }
}
```

The `getLocalPeerInfo` method provided by the `PeerInfoService` interface allows a developer to retrieve cached status information. Unlike `getRemotePeerInfo`, the method returns an `Enumeration` of matching `PeerInfoResponseMessage` s retrieved from the cache. Because this is a local request, registered implementations of `PeerInfoListener` are never invoked as a result of calling the `getLocalPeerInfo` method.

Unlike `getRemotePeerInfo`, `getLocalPeerInfo` can be passed a `null` Peer ID `String`. Passing `null` to `getLocalPeerInfo` returns all the peer information in the local cache.

To run the example, modify the `test.txt` script to use `example7_2` instead of `example7_1`. When run, the output produced by `example7_2` should be roughly the same as `example7_1`, with the distinction that the information is from the cache.

Summary

In its current state, the PIP isn't especially useful. However, work is under way by the Project JXTA team to augment the PIP to provide a more generic status-monitoring framework. It's not clear what features will emerge from this work. Currently, it appears that the PIP implementation will be augmented to provide a way to handle the content of the `request` element sent in the Peer Info Query Message to a remote peer and generate content for the `response` element in the Peer Info Response Message returned by the remote peer. This work will undoubtedly build on the existing classes, so it has been important in this chapter to understand the current PIP, if only to prepare for the arrival of these additional features.

Now that you've seen how the Peer Information Protocol allows a peer to monitor a remote peer, the next chapter explores another mechanism used to transport data between peers: pipes. The Pipe Binding Protocol described in the next chapter is used to establish pipe connections between peers. After a connection is established, pipes allow peers to send data across a virtual connection, abstracting the network transport layer in a generic fashion.

Chapter 8. The Pipe Binding Protocol

Pipes are constructs within JXTA that send data to or receive data from a remote peer. Services typically use either a Resolver handler (refer to Chapter 5, "The Peer Resolver Protocol") or a pipe to communicate with another peer. Before a pipe can actually be used, it must be bound to a peer endpoint. Binding a pipe to an endpoint allows the peers to create either an input pipe for receiving data or an output pipe for sending data. The process of binding a pipe to an endpoint is defined by the Pipe Binding Protocol (PBP).

This chapter explains the Pipe Binding Protocol that JXTA peers use to bind a pipe to an endpoint. The PBP defines a set of messages that a peer can use to query remote peers to find an appropriate endpoint for a given Pipe Advertisement and respond to binding queries from other peers. After a pipe has been bound to an endpoint, a peer can use it to send or receive messages. Several examples in the section "The Pipe Service" demonstrate the use of both input and output pipes to send and receive data, respectively.

Introducing the Pipe Binding Protocol

The *endpoint* is the bottom-most element in the network transport abstraction defined by JXTA. Endpoints are encapsulations of the native network interfaces provided by a peer. These network interfaces typically provide access to low-level transport protocols such as TCP or UDP, although some can provide access to higher-level transport protocols such as HTTP. Endpoints are responsible for producing, sending, receiving, and consuming messages sent across the network. Other services in JXTA build on endpoints either directly or indirectly to provide network connectivity. The Resolver service, for example, builds directly on endpoints, whereas the Discovery service builds on endpoints indirectly via the Resolver service.

In addition to the Resolver service, JXTA offers another mechanism by which services can access a network transport without interacting directly with the endpoint abstraction: pipes. *Pipes* are an abstraction in JXTA that describe a connection between a sending endpoint and one or more receiving endpoints. A pipe is a convenience method layered on top of the endpoint abstraction. Although pipes might appear to provide access to a network transport, implementations of the endpoint abstraction are responsible for the actual task of sending and receiving data over the network.

To provide an abstraction that can encompass the simplest networking technology, JXTA specifies pipes as *unidirectional*, meaning that data travels in only one direction. Pipes are also *asynchronous*, meaning that data can be sent or received at any time, a feature that allows peers to act independently of other peers without any sort of state synchronization. The JXTA Protocols Specification does specify that other types of pipes (bidirectional, synchronous, or streaming) might exist in JXTA. However, only the unidirectional asynchronous variety of pipe is required by the specification.

Pipes are described by a Pipe Advertisement using the XML shown in Listing 8.1.

Listing 8.1 The Pipe Advertisement XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PipeAdvertisement>
  <Id> . . . </Id>
  <Type> . . . </Type>
  <Name> . . . </Name>
</jxta:PipeAdvertisement>
```

The elements of the Pipe Advertisement provide required information that a peer can use to find and connect to a remote peer:

- **`Id`**— A required element containing an ID that uniquely identifies the pipe. This Pipe ID uses the standard JXTA URN format, as described in JXTA Protocols Specification.

- **`Type`**— A required element containing a description of the type of connection possible using the pipe. Currently, the reference implementation supports `JxtaUnicast`, `JxtaUnicastSecure`, and `JxtaPropagate`. The `JxtaUnicast` type

of pipe provides a basic connection between one sending endpoint and one receiving endpoint. The `JxtaUnicastSecure` type of pipe provides the same functionality as the `JxtaUnicast` type of pipe, except that the connection is secured using the Transport Layer Security (TLS) protocol. The `JxtaPropagate` type of pipe provides a broadcast connection between many sending endpoints and multiple receiving endpoints.

- **`Name`—** An optional element containing a symbolic name for the pipe that can be used to discover the Pipe Advertisement using the Discovery service.

Notice that the Pipe Advertisement seems to be missing one important piece of information: a Peer ID. Pipe Advertisements are defined without specifying a specific peer to allow several peers to provide access to a service using the same Pipe Advertisement. The omission of a Peer ID is the reason that pipes must be resolved using the Pipe Binding Protocol.

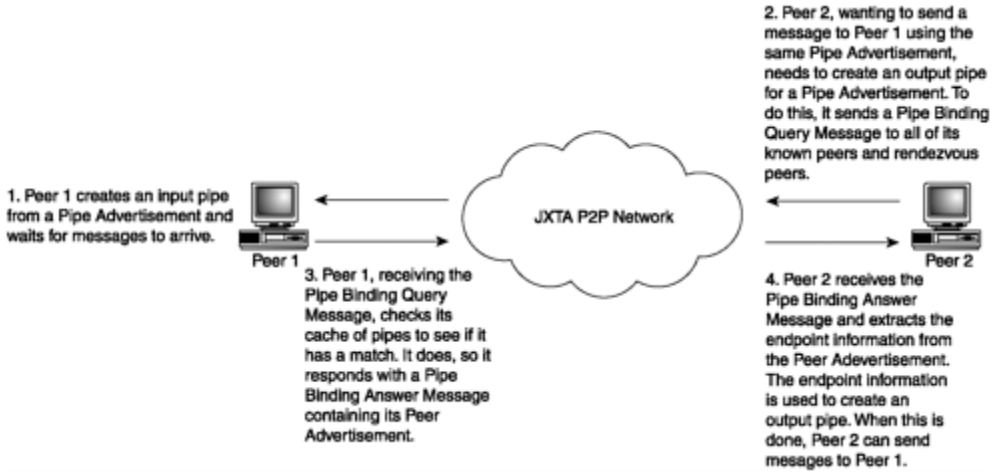
When a peer wants to send data using a pipe, it needs to find a peer that has already bound a pipe with the same Pipe ID to an endpoint and that is listening for data. The PBP defines two messages to enable a peer to resolve a pipe:

- **The Pipe Binding Query Message—** A message format for querying a remote peer if it has bound a pipe with a matching Pipe ID.

- **The Pipe Binding Answer Message—** A message format for sending responses to the query.

The message formats are all that a peer needs to resolve the ID of a peer that has a bound pipe with a given Pipe ID. As shown in Figure 8.1, when a peer wants to bind to a specific pipe, it sends a Pipe Binding Query Message to all of its known peers and rendezvous peers. Peers respond with a Pipe Binding Answer Message that details whether they have a matching bound pipe.

Figure 8.1. Exchange of Pipe Binding Messages.



Two important things should be noted from Figure 8.1. First, when Peer 1 creates an input pipe, nothing is sent to the network. Peer 1 simply begins listening on its local endpoints for incoming messages tagged with the Pipe ID specified in the Pipe Advertisement. Second, the Pipe Advertisement doesn't necessarily need to be communicated over the network. Although a Pipe Advertisement usually is discovered using the Discovery service, a Pipe Advertisement could also be hard-coded into an application or exchanged using the Resolver service.

After the receiving end of the pipe has been resolved to a particular endpoint on a remote peer, the peer can bind the other end of the pipe to its local endpoint. This pipe on the local peer is called an *output pipe* because the pipe has been bound to an endpoint for the purpose of sending output to the remote peer. The bound pipe on the remote peer is called an *input pipe* because the pipe has been bound to an endpoint for the purpose of accepting input. After the sending peer binds an output pipe, it can send messages to the remote peer.

Only the endpoint location of the pipe on a remote peer must be determined in the binding process to create an output pipe. When creating an input pipe, no binding process is necessary because the local peer already knows that it will be binding the Pipe Advertisement to its local endpoint for the purpose of accepting data.

It is important to reiterate that neither the input pipes nor the output pipes are actually responsible for sending or receiving data. The endpoints specified by the bound pipe are the elements responsible for handling the actual exchange of messages over the network.

In the case of propagation pipes (when the Pipe Advertisement's `Type` is set to `JxtaPropagate`), the implementation relies on the multicast or broadcast capabilities of the local endpoint. In this case, the PBP is not required because the sending endpoint doesn't need to find a listening endpoint before it can send data to the network.

The Pipe Binding Query Message

The Pipe Binding Query Message is sent by a peer to resolve the ID of a peer that has bound an input pipe with a specific Pipe ID. Listing 8.2 shows the format of the Pipe Binding Query Message.

Listing 8.2 The Pipe Binding Query Message XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PipeResolver>
  <MsgType>Query</MsgType>
  <PipeId> . . . </PipeId>
  <Type> . . . </Type>
  <Cached> . . . </Cached>
  <Peer> . . . </Peer>
</jxta:PipeResolver>
```

The information sent in the Pipe Binding Query Message describes the pipe that the peer is seeking to resolve and tells whether to use cached information.

- **`MsgType`**— A required element containing a string that indicates the type of Pipe Binding Message. For the Pipe Binding Query Message, this element is hard-coded to `Query`.

- **`PipeId`**— A required element containing the ID of the pipe for which the requesting peer is attempting to resolve a Peer ID.

- **`Type`**— A required element containing the type of pipe being resolved. This corresponds to the `Type` field of the Pipe Advertisement, and it can have a value of `JxtaUnicast`, `JxtaUnicastSecure`, or `JxtaPropagate`.

- **`Cached`**— An optional element that specifies whether the remote peer being queried can use its local cache of resolved pipes to respond to the query. If this parameter is missing, the peer receiving the query assumes that it is allowed to use cached information.

- **`Peer`**— According to the specification, this optional element specifies the Peer ID of the only peer that should respond to the query. However, the current reference implementation does not send this parameter yet; which peers receive the query is specified by the service interface rather than the protocol.

The reference implementation doesn't define any classes to encapsulate the Pipe Binding Query Message.

The Pipe Binding Answer Message

A peer responds to a Pipe Binding Query Message using a Pipe Binding Answer Message. Note that response might or might not be sent to a given query. Responses received are useful only to update the local peer's cached set of resolved pipes. The Pipe Binding Answer Message comes in two forms: one to indicate that a matching pipe was not found and another to indicate a matching pipe was found. Listing 8.3 shows the format of the Pipe Binding Answer Message.

Listing 8.3 The Pipe Binding Answer Message XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PipeResolver>
  <MsgType>Answer</MsgType>
  <PipeId> . . . </PipeId>
  <Type> . . . </Type>
  <Peer> . . . </Peer>
  <Found>false</Found>
  <PeerAdv> . . . </PeerAdv>
</jxta:PipeResolver>
```

The elements of the Pipe Binding Answer Message are nearly identical to those of the Pipe Binding Query Message, with the following exceptions:

- **`MsgType`**— A required element containing a string that indicates the type of Pipe Binding Message. For the Pipe Binding Response Message, this element is hard-coded to `Answer`.

- **`Found`**— An optional element that indicates whether a matching Pipe ID was found. If this element is missing, the reference implementation assumes that a peer with a matching Pipe ID was found.

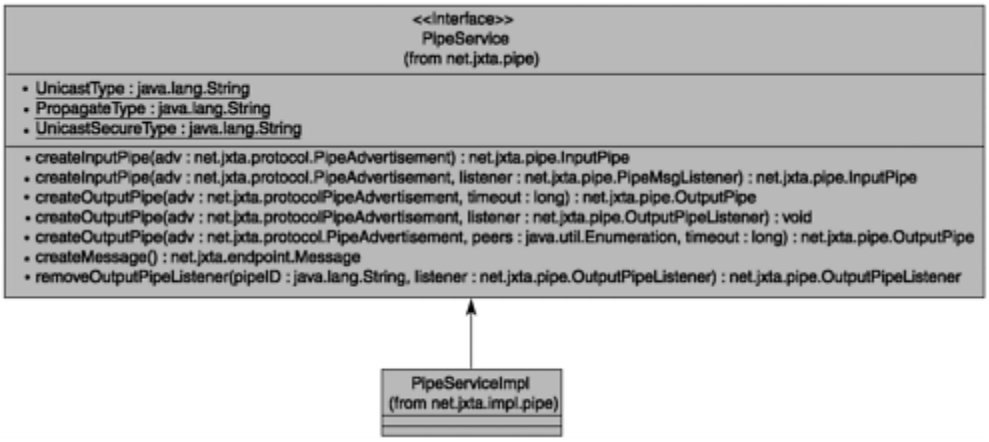
- **`PeerAdv`**— An optional element containing the Peer Advertisement of the peer that has the matching Pipe ID. If no match was found, this element does not appear in the Pipe Binding Answer Message. The endpoint information required to contact a remote peer using a specific pipe is included as part of the Peer Advertisement.

As with the Pipe Binding Query Message, the reference implementation provides no classes to abstract the Pipe Binding Answer Message.

The Pipe Service

As with every other protocol in JXTA, the Pipe Binding Protocol is provided as a service. In the case of the PBP, the Pipe service is responsible for handling the details of creating input or output pipe objects and binding those pipe objects to endpoints. The Pipe service, as shown in Figure 8.2, is defined by the `PipeService` interface in the `net.jxta.pipe` package, with a reference implementation defined by the `PipeServiceImpl` class in the `net.jxta.impl.pipe` package.

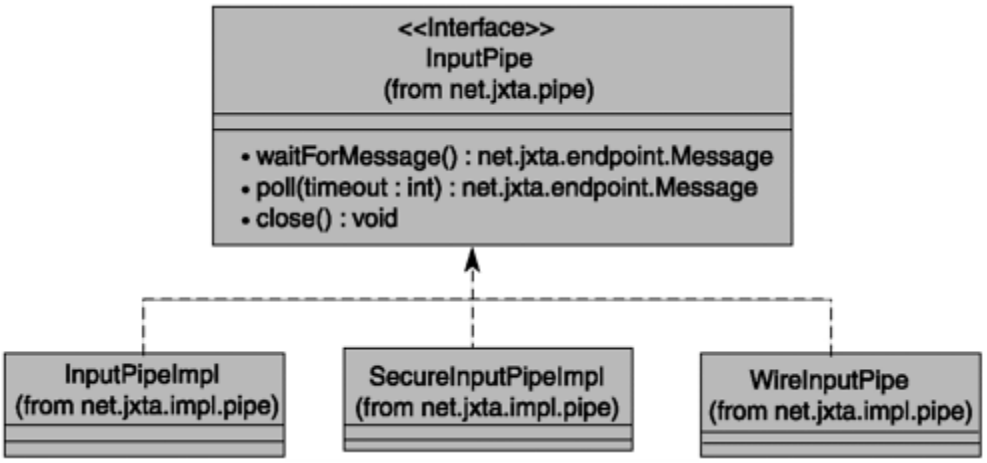
Figure 8.2. The Pipe Service interface and implementation.



One thing that should be noted about the reference implementation of the `PipeService` interface is its reliance on another service to implement the Pipe Binding Protocol. The `PipeResolver` service is a Resolver service handler that provides a convenience mechanism for `PipeServiceImpl`, freeing it to focus on matching resolved endpoints to pipe-implementation objects.

Pipe objects implement either the `InputPipe` or the `OutputPipe` interfaces defined in the `net.jxta.pipe` package. The reference implementation provides an implementation of these interfaces for each of the three types of pipe (unicast, secure unicast, and propagate), as shown in Figure 8.3.

Figure 8.3. The pipe interfaces and classes.



A developer never creates these `InputPipe` or `OutputPipe` implementations directly. Instead, a developer obtains an `InputPipe` or `OutputPipe` instance using `PipeService`'s `createOutputPipe` or `createInputPipe` methods, respectively.

Using the Pipe Service to Send and Receive Messages

The examples in the following sections demonstrate how to use the Pipe service and pipes to send and receive data. The example consists of three parts: an advertisement generator, a client, and a server.

Starting and Stopping the JXTA Platform

Unlike previous examples in this book, the examples in this chapter do not rely on the Shell to start the JXTA platform or provide the user interface. These applications start and stop the JXTA platform themselves by creating a Net Peer Group instance using this call:

```
PeerGroup peerGroup = PeerGroupFactory.newNetPeerGroup();
```

As you've seen in all the examples so far, all operations within JXTA are associated with a peer group. In the examples in all the previous chapters, a `PeerGroup` object obtained from the Shell environment was used to obtain an instance of a core service, such as the Discovery service, for a peer group.

The Net Peer Group is a special peer group, one that is described in greater detail in Chapter 10, "Peer Groups and Services." For the moment, just think of the Net Peer Group as a common peer group that peers belong to when the platform is started.

Unfortunately, after the platform starts, there currently isn't any nice way to shut down the JXTA platform in a controlled way. The only way, as unpleasant as it is, is to use this code:

```
System.exit(0);
```

The `exit` call takes an integer parameter, where `0` indicates no error occurred. To stop the JXTA platform after an error has occurred, the `exit` method should be called with a nonzero value, usually `1`.

Creating a Pipe Advertisement

Creating an input pipe or output pipe using the Pipe service requires a Pipe Advertisement. So, as shown in Listing 8.4, the first step in creating any solution that involves pipes is to create a Pipe Advertisement that describes the type of pipe, the Pipe ID, and an optional name for the pipe.

Listing 8.4 Source Code for *PipeAdvPopulator.java*

```
package com.newriders.jxta.chapter8;

import java.io.FileWriter;
import java.io.IOException;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredTextDocument;

import net.jxta.exception.PeerGroupException;

import net.jxta.id.ID;
import net.jxta.id.IDFactory;

import net.jxta.impl.peergroup.StdPeerGroup;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;

import net.jxta.pipe.PipeService;

import net.jxta.protocol.PipeAdvertisement;
```

```
/**
 * An example to create a set of common Pipe Advertisement to be used
 * by the PipeServer example application.
 */
public class PipeAdvPopulator
{
    /**
     * The peerGroup for the application.
     */
    private PeerGroup peerGroup = null;
    /**
     * Generates a Pipe Advertisement for the PipeClient/Server example.
     */
    public void generatePipeAdv()
    {
        // Create a new Pipe Advertisement object instance.
        PipeAdvertisement pipeAdv =
            (PipeAdvertisement) AdvertisementFactory.newAdvertisement(
                PipeAdvertisement.getAdvertisementType());

        // Create a unicast Pipe Advertisement.
        pipeAdv.setName("Chapter 8 Example Unicast Pipe Advertisement");
        pipeAdv.setPipeID((ID) IDFactory.newPipeID(
            peerGroup.getPeerGroupID()));
        pipeAdv.setType(PipeService.UnicastType);
        writePipeAdv(pipeAdv, "UnicastPipeAdv.xml");

        // Create a secure unicast Pipe Advertisement.
        pipeAdv.setName(
            "Chapter 8 Example Secure Unicast Pipe Advertisement");
        pipeAdv.setPipeID((ID) IDFactory.newPipeID(
            peerGroup.getPeerGroupID()));
        pipeAdv.setType(PipeService.UnicastSecureType);
        writePipeAdv(pipeAdv, "SecureUnicastPipeAdv.xml");
```

```
    // Create a propagate Pipe Advertisement.
    pipeAdv.setName("Chapter 8 Example Propagate Pipe Advertisement");
    pipeAdv.setPipeID((ID) IDFactory.newPipeID(
        peerGroup.getPeerGroupID()));
    pipeAdv.setType(PipeService.PropagateType);
    writePipeAdv(pipeAdv, "PropagatePipeAdv.xml");
}

/**
 * Starts the JXTA platform.
 *
 * @exception   PeerGroupException thrown if the platform can't
 *              be started.
 */
public void initializeJXTA() throws PeerGroupException
{
    peerGroup = PeerGroupFactory.newNetPeerGroup();
}

/**
 * Runs the application: starts the JXTA platform, generates the Pipe
 * Advertisements, and stops the JXTA platform.
 *
 * @param   args the command-line arguments passed to the application.
 */
public static void main(String[] args)
{
    PipeAdvPopulator p = new PipeAdvPopulator();

    try
    {
        // Initialize the JXTA platform.
        p.initializeJXTA();

        // Generate the Pipe Advertisements to be used by the examples.
```

```
            p.generatePipeAdv();

            // Stop the JXTA platform.
            p.uninitializeJXTA();
        }
        catch (PeerGroupException e)
        {
            System.out.println("Error starting JXTA platform: " + e);
            System.exit(1);
        }
    }

    /**
     * Stops the JXTA platform.
     */
    public void uninitializeJXTA()
    {
        // Currently, there isn't any nice way to do this.
        System.exit(0);
    }

    /**
     * Writes the given Pipe Advertisement to a file
     * with the specified name.
     *
     * @param   pipeAdv the Pipe Advertisement to be written to file.
     * @param   fileName the name of the file to write.
     */
    private void writePipeAdv(PipeAdvertisement pipeAdv, String fileName)
    {
        // Create an XML formatted version of the Pipe Advertisement.
        try
        {
            FileWriter file = new FileWriter(fileName);
            MimeMediaType mimeType = new MimeMediaType("text/xml");
            StructuredTextDocument document =
```

```
                (StructuredTextDocument) pipeAdv.getDocument(mimeType);

            // Output the XML for the advertisement to the file.
            document.sendToWriter(file);
            file.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }
    }
}
```

The `PipeAdvPopulator` example creates a Pipe Advertisement for each possible pipe type, to allow you to experiment with all the pipe types in the following pipe examples. `PipeAdvPopulator` creates three files: `UnicastPipeAdv.xml`, `SecureUnicastPipeAdv.xml`, and `PropagatePipeAdv.xml`.

To compile and run `PipeAdvPopulator`, create a new directory and copy into it all the `JAR` files from the `lib` directory under the JXTA Demo install directory. Place `PipeAdvPopulator.java` in the same directory and compile it from the command line using this code:

```
javac -d . -classpath .;beepcore.jar;cms.jar;cryptix32.jar;
cryptix-asn1.jar;instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;
jxtashell.jar;log4j.jar;minimalBC.jar PipeAdvPopulator.java
```

Run the example using this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;
cryptix-asn1.jar;instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;
jxtashell.jar;log4j.jar;minimalBC.jar
com.newriders.jxta.chapter8.PipeAdvPopulator
```

Configure your peer as you did for the earlier Shell examples when prompted by the configuration screens. When you finish the configuration, the JXTA platform starts and `PipeAdvPopulator` creates the Pipe Advertisement files.

Creating an Input Pipe

An input pipe listens for messages being sent by other peers. The example in Listing 8.5 creates an `InputPipe` instance using the Pipe service.

Listing 8.5 Source Code for *PipeServer.java*

```
package com.newriders.jxta.chapter8;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import java.awt.FlowLayout;
import java.awt.Container;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;

import net.jxta.discovery.DiscoveryService;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;

import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;
```

```
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.PipeMsgEvent;
import net.jxta.pipe.PipeMsgListener;
import net.jxta.pipe.PipeService;

import net.jxta.protocol.PipeAdvertisement;


/**
 * A server application to accept incoming messages on a pipe and display
 * them to the user.
 */
public class PipeServer implements PipeMsgListener
{
    /**
     * The frame for the user interface.
     */
    private JFrame serverFrame = new JFrame("PipeServer");

    /**
     * A label used to display the received message in the GUI.
     */
    private JLabel messageText =
        new JLabel("Waiting to receive a message...");

    /**
     * The peerGroup for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * Indicates whether the GUI has been initialized already.
     */
```

```
private boolean initialized = false;

/**
 * The input pipe used to receive messages.
 */
private InputPipe inputPipe = null;
/**
 * Starts the JXTA platform.
 *
 * @exception   PeerGroupException thrown if the platform can't
 *              be started.
 */
public void initializeJXTA() throws PeerGroupException
{
    peerGroup = PeerGroupFactory.newNetPeerGroup();
}

/**
 * Load the Pipe Advertisement generated by PipeAdvPopulator.
 * This method tries to create an output pipe that can be used
 * to send messages.
 *
 * @param      fileName the name of the file from which to load
 *             the Pipe Advertisement.
 * @exception   FileNoteFoundException if the Pipe Advertisement
 *             file can't be found.
 * @exception   IOException if there is an error binding the pipe.
 */
public void loadPipeAdv(String fileName)
    throws FileNotFoundException, IOException
{
    FileInputStream file = new FileInputStream(fileName);
    MimeMediaType asMimeType = new MimeMediaType("text/xml");

    // Load the advertisement.
    PipeAdvertisement pipeAdv =
```

```
            (PipeAdvertisement) AdvertisementFactory.newAdvertisement(
                asMimeType, file);

        // Publish the discovery to allow peers to find and bind the pipe.
        DiscoveryService discovery = peerGroup.getDiscoveryService();
        discovery.publish(pipeAdv, DiscoveryService.ADV);
        discovery.remotePublish(pipeAdv, DiscoveryService.ADV);

        // Create an input pipe using the advertisement.
        PipeService pipeService = peerGroup.getPipeService();
        inputPipe = pipeService.createInputPipe(pipeAdv, this);
    }
    /**
     * Runs the application: starts the JXTA platform, loads the
     * Pipe Advertisement from file, and creates an input pipe to
     * use to receive messages.
     *
     * @param   args the command-line arguments passed to the application.
     */
    public static void main(String[] args)
    {
        PipeServer server = new PipeServer();

        if (args.length == 1)
        {
            try
            {
                // Initialize the JXTA platform.
                server.initializeJXTA();

                // Load the Pipe Advertisement.
                server.loadPipeAdv(args[0]);

                // Show the user interface.
                server.showGUI();
            }
```

```
        catch (PeerGroupException e)
        {
            System.out.println("Error starting JXTA platform: " + e);
            System.exit(1);
        }
        catch (FileNotFoundException e2)
        {
            System.out.println("Unable to load Pipe Advertisement: "
                + e2);
            System.exit(1);
        }
        catch (IOException e3)
        {
            System.out.println("Error loading Pipe Advertisement: "
                + e3);
            System.exit(1);
        }
    }
    else
    {
         System.out.println(
             "Specify the name of the input Pipe Advertisement file.");
    }
}

/**
 * Handles an incoming message.
 *
 * @param   event the incoming event containing the arriving message.
 */
public void pipeMsgEvent(PipeMsgEvent event)
{
    // Extract the message.
    Message message = event.getMessage();

    // Set the user interface to display the message text.
```

```
    messageText.setText(message.getString("MessageText"));
}

/**
 * Configures and displays a simple user interface to display messages
 * received by the pipe. The GUI also allows the user to stop the
 * server application.
 */
public void showGUI()
{
    if (!initialized)
    {
        initialized = true;

        JButton quitButton = new JButton("Quit");

        // Populate the GUI frame.
        Container pane = serverFrame.getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(messageText);
        pane.add(quitButton);

        quitButton.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    serverFrame.hide();

                    // Stop the JXTA platform. Currently, there isn't
                    // any nice way to do this.
                    System.exit(0);
                }
            }
        );
```

```
            // Pack and display the user interface.
            serverFrame.pack();
            serverFrame.show();
        }
    }
}
```

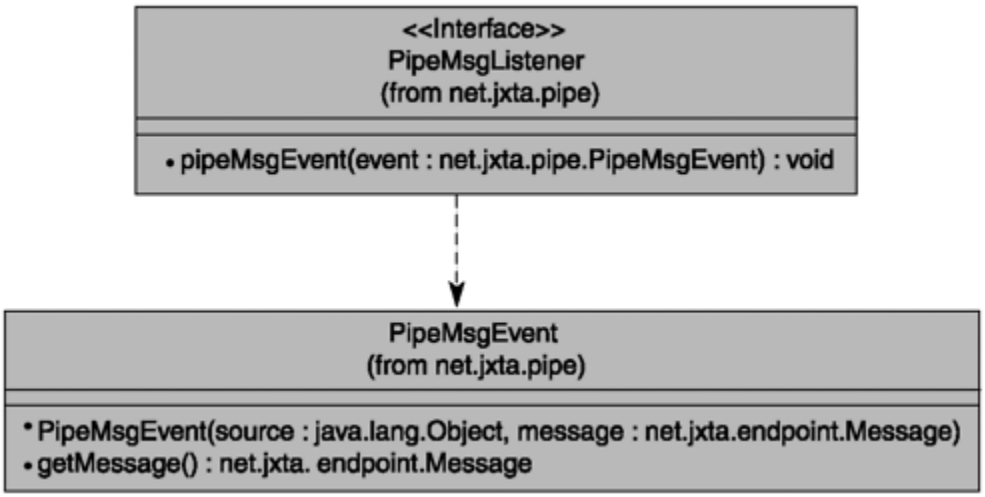
The `PipeServer` starts the JXTA platform, loads a Pipe Advertisement specified at the command line, creates an input pipe from the advertisement, and waits for messages to arrive that it can display in its user interface.

The `PipeServer` example creates an input pipe using this code:

```
inputPipe = pipeService.createInputPipe(pipeAdv, this);
```

As shown in Figure 8.4, this version of `createInputPipe` takes `PipeMsgListener` as its second parameter. The `PipeServer` class itself implements the `PipeMsgListener` interface to receive notification when new messages arrive through the newly created `InputPipe`.

Figure 8.4. The `PipeMsgListener` interface and `PipeMsgEvent` class.



This is the only mechanism for an application to register a listener because the `InputPipe` interface doesn't define any methods to register or unregister a listener object. The

`PipeServer` example implements the `PipeMsgListener` 's `pipeMsgEvent` method to extract the received `Message` and update the `PipeServer` user interface.

An application that doesn't use a `PipeMsgListener` can still retrieve messages received by `InputPipe` by using either the `poll` or `waitForMessage` methods defined by `InputPipe`:

```
public Message poll(int timeout) throws InterruptedException;

public Message waitForMessage() throws InterruptedException;
```

The `waitForMessage` method blocks indefinitely until a message arrives, at which point, it returns a `Message` object. Usually an application that wants to use this method spawns its own subclass of `Thread` to handle calling `waitForMessage` repeatedly and processing the `Message` objects as they arrive.

The `poll` method is similar to `waitForMessage`, except that a call to the `poll` method blocks only for the length of time specified. The `timeout` argument specifies the amount of time (in milliseconds) to wait for a `Message` to arrive before returning. If no message is received, the `poll` method returns `null`.

By itself, the `PipeServer` example isn't very useful. There's no point waiting for messages to arrive if no one's sending messages! Before you can use `PipeServer`, you need to create a client application that sends messages using the same Pipe Advertisement.

Creating an Output Pipe

An output pipe sends messages to a remote peer. The example in Listing 8.6 creates an output pipe to send simple text messages to a peer running the `PipeServer` example created in the previous section.

Listing 8.6 Source Code for *PipeClient.java*

```
package com.newriders.jxta.chapter8;

import java.awt.FlowLayout;
```

```
import java.awt.Container;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import java.net.URL;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JTextField;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;

import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;

import net.jxta.id.IDFactory;

import net.jxta.peer.PeerID;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;

import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.OutputPipeEvent;
import net.jxta.pipe.OutputPipeListener;
import net.jxta.pipe.PipeService;

import net.jxta.protocol.PipeAdvertisement;
```

```
/**
 * A client application, which sends messages over a pipe to a remote peer.
 */
public class PipeClient implements OutputPipeListener
{
    /**
     * The peerGroup for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * The pipe to use to send the message to the remote peer.
     */
    private OutputPipe outputPipe = null;

    /**
     * The frame for the user interface.
     */
    private JFrame clientFrame = new JFrame("PipeClient");

    /**
     * The text field in the user interface to accept the message
     * text to be sent over the pipe.
     */
    private JTextField messageText = new JTextField(20);

    /**
     * Indicates whether the pipe has been bound already.
     */
    private boolean initialized = false;


    /**
     * Starts the JXTA platform.
     *
     * @exception   PeerGroupException thrown if the platform can't
```

```
     *              be started.
     */
    public void initializeJXTA() throws PeerGroupException
    {
        peerGroup = PeerGroupFactory.newNetPeerGroup();
    }
    /**
     * Load the Pipe Advertisement generated by PipeAdvPopulator. This
method
     * tries to create an output pipe that can be used to send messages.
     *
     * @param       fileName the name of the file from which to load
     *              the Pipe Advertisement.
     * @exception   FileNoteFoundException if the Pipe Advertisement file
     *              can't be found.
     * @exception   IOException if there is an error binding the pipe.
     */
    public void loadPipeAdv(String fileName)
        throws FileNotFoundException, IOException
    {
        FileInputStream file = new FileInputStream(fileName);
        MimeMediaType asMimeType = new MimeMediaType("text/xml");

        // Load the advertisement.
        PipeAdvertisement pipeAdv =
            (PipeAdvertisement) AdvertisementFactory.newAdvertisement(
                asMimeType, file);

        // Create an output pipe using the advertisement. This version of
        // createOutputPipe uses the PipeClient class as an
        // OutputPipeListener object.
        PipeService pipeService = peerGroup.getPipeService();
        pipeService.createOutputPipe(pipeAdv, this);
    }

    /**
```

```
 * Runs the application: starts the JXTA platform, loads the Pipe
 * Advertisement from file, and attempts to resolve the pipe.
 *
 * @param   args the command-line arguments passed to the application.
 */
public static void main(String[] args)
{
    PipeClient client = new PipeClient();

    if (args.length == 1)
    {
        try
        {
            // Initialize the JXTA platform.
            client.initializeJXTA();

            // Load the Pipe Advertisement.
            client.loadPipeAdv(args[0]);
        }
        catch (PeerGroupException e)
        {
            System.out.println("Error starting JXTA platform: " + e);
            System.exit(1);
        }
        catch (FileNotFoundException e2)
        {
            System.out.println("Unable to load Pipe Advertisement: "
                + e2);
            System.exit(1);
        }
        catch (IOException e3)
        {
            System.out.println("Error loading or binding Pipe"
                + " Advertisement: " + e3);
            System.exit(1);
        }
```

```
        }
        else
        {
            System.out.println("You must specify the name of the input"
                + " Pipe Advertisement file.");
        }
    }

    /**
     * The OutputPipeListener event that is triggered when an OutputPipe
is
     * resolved by the call to PipeService.createOutputPipe.
     *
     * @param   event the event to use to extract the resolved output pipe.
     */
    public void outputPipeEvent(OutputPipeEvent event)
    {
        // We care about only the first pipe we manage to resolve.
        if (!initialized)
        {
            initialized = true;

            // Get the bound pipe.
            outputPipe = event.getOutputPipe();

            // Show a small GUI to allow the user to send a message.
            showGUI();
        }
    }

    /**
     * Sends a message string to the remote peer using the output pipe.
     *
     * @param   messageString the message text to send to the remote peer.
     */
    private void sendMessage(String messageString)
```

```
    {
        PipeService pipeService = peerGroup.getPipeService();
        Message message = pipeService.createMessage();

        // Configure the message object.
        message.setString("MessageText", messageString);

        if (null != outputPipe)
        {
            try
            {
                // Send the message.
                outputPipe.send(message);
            }
            catch (IOException e)
            {
                // Show some warning dialog.
                JOptionPane.showMessageDialog(null, e.toString(),
"Error",
                    JOptionPane.WARNING_MESSAGE);
            }
        }
        else
        {
            // Show some warning dialog.
            JOptionPane.showMessageDialog(null, "Output pipe is null!",
                "Error", JOptionPane.WARNING_MESSAGE);
        }
    }

    /**
     * Configures and displays a simple user interface to allow the user
to
     * send text messages. The GUI also allows the user to stop the client
     * application.
     */
```

```
private void showGUI()
{
    JButton sendButton = new JButton("Send Message");
    JButton quitButton = new JButton("Quit");

    // Populate the GUI frame.
    Container pane = clientFrame.getContentPane();
    pane.setLayout(new FlowLayout());
    pane.add(messageText);
    pane.add(sendButton);
    pane.add(quitButton);

    // Set up listeners for the buttons.
    sendButton.addActionListener(
        new ActionListener() {
            public void actionPerformed(ActionEvent e)
            {
                // Send the message.
                sendMessage(messageText.getText());

                // Clear the text.
                messageText.setText("");
            }
        }
    );
    quitButton.addActionListener(
        new ActionListener()
        {
            public void actionPerformed(ActionEvent e)
            {
                clientFrame.hide();

                // Stop the JXTA platform. Currently, there isn't any
                // nice way to do this.
                System.exit(0);
            }
```

```
            }
        );

        // Pack and display the user interface.
        clientFrame.pack();
        clientFrame.show();
    }
}
```

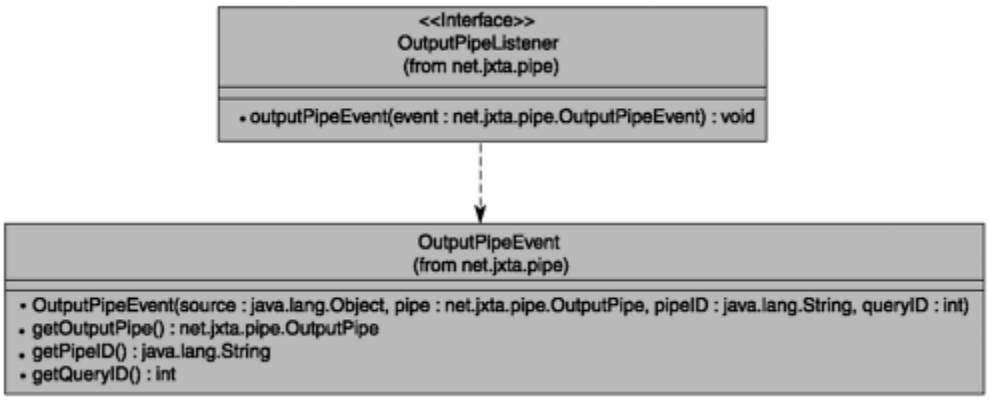
The `PipeClient` example mirrors the `PipeServer` example. The `PipeClient` example starts the JXTA platform, loads a Pipe Advertisement specified at the command line, and creates an output pipe from the advertisement. After an output pipe is successfully created, the example displays a user interface that the user can use to input messages to be sent via the output pipe to a remote peer.

The `PipeClient` example creates an output pipe using this code:

```
pipeService.createOutputPipe(pipeAdv, this);
```

This version of `createOutputPipe` takes `OutputPipeListener`, shown in Figure 8.5, as its second parameter. The `PipeClient` itself implements the `OutputPipeListener` interface to receive notification when an output pipe has been successfully created.

Figure 8.5. The `OutputPipeListener` interface and `OutputPipeEvent` class.



Unlike the `PipeServer` example, the `PipeClient` example doesn't display its user interface immediately. Instead, `PipeClient` 's implementation of `OutputPipeListener` 's `outputPipeEvent` method displays the user interface when a pipe has been bound to an endpoint successfully. Because an output pipe may be bound successfully to several endpoints, `outputPipeEvent` does this only the first time it is called. Text entered into the user interface is wrapped as a `Message` and sent over the resolved `OutputPipe` using `OutputPipe` 's `send` method.

Using PipeServer and PipeClient

`PipeServer` and `PipeClient` each form one end of a complete communication connection. The `PipeServer` class listens for data on an input pipe, and the `PipeClient` class allows a user to send data using an output pipe. To prepare to run these examples, follow these steps:

1. Place the source code in the same directory that you created for the `PipeAdvPopulator` example.

2. Compile the source code by using the same command as before (replacing `PipeAdvPopulator.java` with the appropriate source filename, of course).

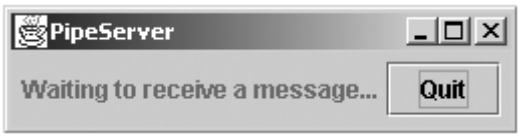
3. Create a copy of the entire directory. This is required so that you can run two independent instances of the `PipeServer` and `PipeClient` applications.

Next, start the `PipeServer` example in the original directory using this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar com.newriders.jxta.chapter8.PipeServer
UnicastPipeAdv.xml
```

Here, the `UnicastPipeAdv.xml` parameter specifies that `PipeServer` should use the `UnicastPipeAdv.xml` Pipe Advertisement file to create the input pipe. After the input pipe is created, the `PipeServer` example displays the user interface in Figure 8.6.

Figure 8.6. The `PipeServer` user interface.



Finally, start `PipeClient` in the copy of the original directory. For this to work, you need to force the JXTA platform to show the configuration interface by deleting the PlatformConfig file. Start `PipeClient` using this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar com.newriders.jxta.chapter8.PipeClient
UnicastPipeAdv.xml
```

When the configuration screen appears, choose a different TCP and HTTP port in the TCP and HTTP Settings sections of the Advanced tab. After you enter the configuration and started the platform, `PipeClient` attempts to bind an output pipe. When the output pipe has been successfully bound, `PipeClient` displays the user interface in Figure 8.7.

Figure 8.7. The `PipeClient` user interface.

You should now be able to enter message in `PipeClient` 's user interface and send it by clicking Send Message. The message is sent using the output pipe, and the `PipeServer` user interface displays the message.

Note that although this demonstration might make it appear as though the communication between the client and the server is reliable, JXTA does not guarantee message delivery. Even if the pipe is using an endpoint protocol built on top of a reliable network transport, such as TCP, a message is not guaranteed to be delivered. A message might be dropped en route by an overloaded intermediary peer or even by the destination peer itself. That said,

reliable message delivery could be built on top of pipes fairly easily and will most likely be included in JXTA in the future.

Using Secure Pipes

A JXTA application can easily switch to using secure pipes just by changing the Pipe Advertisement used when creating the input and output pipes. To try using `PipeServer` and `PipeClient` with secure pipes, start the application the same way as in the previous section, but replace `UnicastPipeAdv.xml` in each command with `SecureUnicastPipeAdv.xml`.

Secure pipes use the Transport Security Layer protocol, a variant of SSL 3.0, to secure the communication channel. When you configure the platform for the first time, the platform generates a root certificate and private key that are used to secure communications. The root certificate is saved in the Personal Security Environment directory (`pse`) under the current directory when the platform executes, and the private key is protected using the password entered in the Security tab of the Configurator. The root certificate is also published within the Peer Advertisement.

Using secure pipes with `PipeServer` and `PipeClient` should not seem any different than using the nonsecure unicast pipes in the previous example.

Using Propagation Pipes

Propagation pipes are different than the other two types of pipes examined so far in this chapter. Propagation pipes provide a peer with a convenient mechanism to broadcast data to multiple peer endpoints. This might be useful in some applications, such as a chat application, in which one peer produces data for consumption by multiple remote peers.

In theory, you can use a propagation pipe by invoking `PipeClient` and `PipeServer` using the `PropagatePipeAdv.xml` Pipe Advertisement instead of the `UnicastPipeAdv.xml`. However, the current reference implementation of `PipeService` does not allow you to call `createOutputPipe` and provide an `OutputPipeListener`. This should be fixed shortly, but in case it isn't, Listing 8.7 shows a modified version of `PipeClient` that fixes the problem.

Listing 8.7 Source Code for *PropagatePipeClient.java*

```
package com.newriders.jxta.chapter8;

import java.awt.FlowLayout;
import java.awt.Container;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import java.net.URL;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JOptionPane;
import javax.swing.JTextField;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;

import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;

import net.jxta.id.IDFactory;

import net.jxta.peer.PeerID;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;

import net.jxta.pipe.OutputPipe;
```

```
import net.jxta.pipe.PipeService;

import net.jxta.protocol.PipeAdvertisement;


/**
 * A client application, which sends messages over a pipe to a remote peer.
 * This version is slightly different, to allow for use of a propagation
 * pipe.
 */
public class PropagatePipeClient
{
    /**
     * The peerGroup for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * The pipe to use to send the message to the remote peer.
     */
    private OutputPipe outputPipe = null;
    /**
     * The frame for the user interface.
     */
    private JFrame clientFrame = new JFrame("PropagatePipeClient");

    /**
     * The text field in the user interface to accept the message
     * text to be sent over the pipe.
     */
    private JTextField messageText = new JTextField(20);


    /**
     * Starts the JXTA platform.
     *
```

```
     * @exception   PeerGroupException thrown if the platform can't
     * be started.
     */
    public void initializeJXTA() throws PeerGroupException
    {
        peerGroup = PeerGroupFactory.newNetPeerGroup();
    }

    /**
     * Load the Pipe Advertisement generated by PipeAdvPopulator. This
     * method tries to create an output pipe that can be used to send
messages.
     *
     * @param        fileName the name of the file from which to load the
     *               Pipe Advertisement.
     * @exception    FileNoteFoundException if the Pipe Advertisement
     *               file can't be found.
     * @exception    IOException if there is an error binding the pipe.
     */
    public void loadPipeAdv(String fileName)
        throws FileNotFoundException, IOException
    {
        FileInputStream file = new FileInputStream(fileName);
        MimeMediaType asMimeType = new MimeMediaType("text/xml");

        // Load the advertisement.
        PipeAdvertisement pipeAdv =
            (PipeAdvertisement) AdvertisementFactory.newAdvertisement(
                asMimeType, file);

         // Create an output pipe using the advertisement. This version of
         // createOutputPipe uses the PipeClient class as an
         // OutputPipeListener object.
         PipeService pipeService = peerGroup.getPipeService();
         outputPipe = pipeService.createOutputPipe(pipeAdv, 10000);
```

```
            // Because we can't use an OutputPipeListener when attempting to
            // create an output propagation pipe, the GUI must be displayed
            // immediately.
        showGUI();
    }

    /**
     * Runs the application: starts the JXTA platform, loads the Pipe
     * Advertisement from file, and attempts to resolve the pipe.
     *
     * @param   args the command-line arguments passed to the application.
     */
    public static void main(String[] args)
    {
        PropagatePipeClient client = new PropagatePipeClient();

        if (args.length == 1)
        {
            try
            {
                // Initialize the JXTA platform.
                client.initializeJXTA();

                // Load the Pipe Advertisement.
                client.loadPipeAdv(args[0]);
            }
            catch (PeerGroupException e)
            {
                System.out.println("Error starting JXTA platform: " + e);
                System.exit(1);
            }
            catch (FileNotFoundException e2)
            {
                System.out.println("Unable to load Pipe Advertisement: "
                    + e2);
                System.exit(1);
```

```
        }
        catch (IOException e3)
        {
            System.out.println("Error loading or binding Pipe"
                + " Advertisement: " + e3);
            System.exit(1);
        }
    }
    else
    {
        System.out.println("You must specify the name of the input"
            + " Pipe Advertisement file.");
    }
}

/**
 * Sends a message string to the remote peer using the output pipe.
 *
 * @param   messageString the message text to send to the remote peer.
 */
private void sendMessage(String messageString)
{
    PipeService pipeService = peerGroup.getPipeService();
    Message message = pipeService.createMessage();

    // Configure the message object.
    message.setString("MessageText", messageString);

    if (null != outputPipe)
    {
        try
        {
            // Send the message.
            outputPipe.send(message);
         }
         catch (IOException e)
```

```
             {
                // Show some warning dialog.
                JOptionPane.showMessageDialog(null, e.toString(),
"Error",
                    JOptionPane.WARNING_MESSAGE);
            }
        }
        else
        {
            // Show some warning dialog.
            JOptionPane.showMessageDialog(null, "Output pipe is null!",
                "Error", JOptionPane.WARNING_MESSAGE);
        }
    }

    /**
     * Configures and displays a simple user interface to allow the user
to
     * send text messages. The GUI also allows the user to stop the client
     * application.
     */
    private void showGUI()
    {
        JButton sendButton = new JButton("Send Message");
        JButton quitButton = new JButton("Quit");

        // Populate the GUI frame.
        Container pane = clientFrame.getContentPane();
        pane.setLayout(new FlowLayout());
        pane.add(messageText);
        pane.add(sendButton);
        pane.add(quitButton);

        // Set up listeners for the buttons.
        sendButton.addActionListener(
            new ActionListener() {
```

```
                public void actionPerformed(ActionEvent e)
                {
                    // Send the message.
                    sendMessage(messageText.getText());

                    // Clear the text.
                    messageText.setText("");
                }
            }
        );
        quitButton.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    clientFrame.hide();

                    // Stop the JXTA platform. Currently, there isn't any
                    // nice way to do this.
                    System.exit(0);
                }
            }
        );

        // Pack and display the user interface.
        clientFrame.pack();
        clientFrame.show();
    }
}
```

Instead of calling `createOutputPipe` in `loadPipeAdv` with an `OutputPipeListener` object, this version calls `createOutputPipe` with a timeout value. The user interface is shown in `loadPipeAdv` after the output pipe is bound rather than being shown by the `outputPipeEvent` method.

To see the propagate pipe in action, create another copy of the directory holding your source code and the JXTA JARs, and delete the `PlatformConfig` file. This time, run two `PipeServer` instances from different directories, and run a `PropagatePipeClient` instance from third directory. Remember to configure the platform running in the newest directory (the one that you copied at the beginning of this paragraph) to use another TCP and HTTP port as before. When running these applications, also be sure to use the `PropagatePipeAdv.xml` file as the source of the Pipe Advertisement.

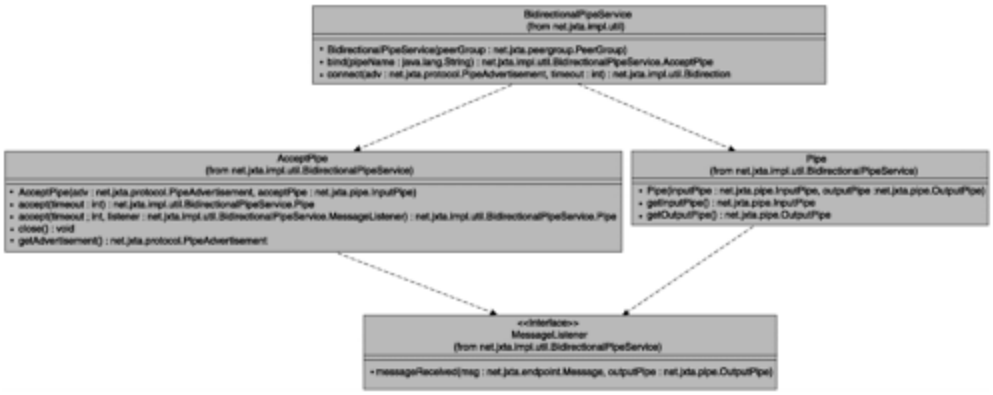
When the `PipeServer` and `PropagatePipeClient` instances are running, you should be able to send a message using `PropagatePipeClient`. When the message is sent, it should be displayed by both `PipeServer` instances. By comparison, performing the same exercise using the `UnicastPipeAdv.xml` Pipe Advertisement would result in only one of the `PipeServer` instances receiving the message. In this case, only the first pipe instance resolved by the Pipe service would receive the message.

Bidirectional Pipes

The examples given so far in this chapter have demonstrated only unidirectional communication. To achieve bidirectional communication, you need two pipes: one to send data and one to receive data.

You can easily implement a bidirectional solution, but doing so requires you to write the code to bind both the input and output pipes. Instead of writing the code, you can use the `BidirectionalPipeService` class, shown in Figure 8.8, from the `net.jxta.impl.util` package to handle the common tasks of initializing pipes for two-way communications.

Figure 8.8. The `BidirectionalPipeService` class and supporting classes.

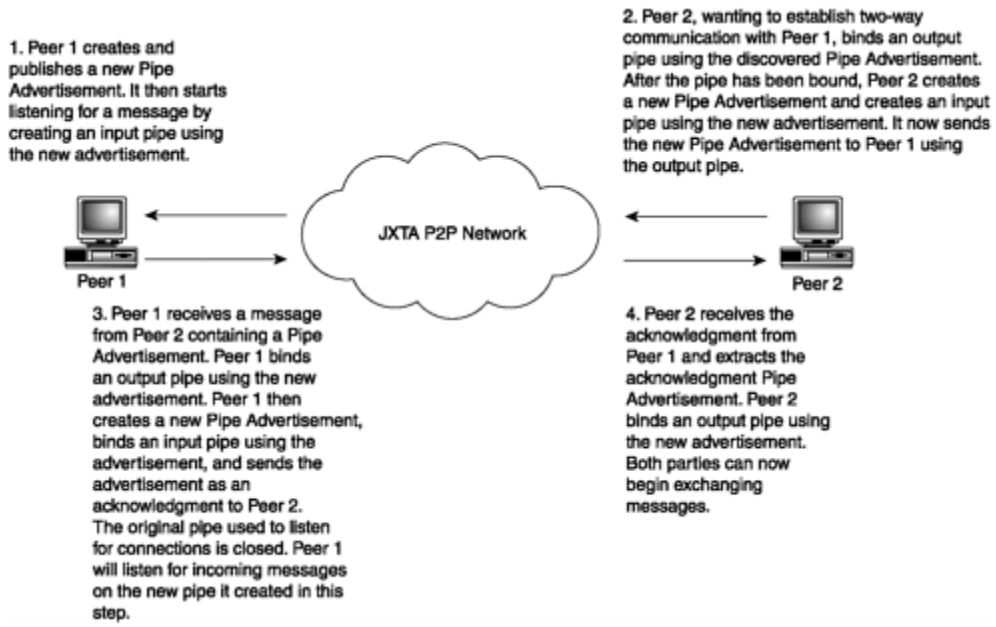


The `BidirectionalPipeService` class provided by the reference implementation isn't a real service. Unlike `PipeService` or any of the other core services, `BidirectionalPipeService` is not constantly running on a peer waiting to handle incoming messages. Instead, `BidirectionalPipeService` is simply a wrapper built on top of the Pipe and Discovery services. `BidirectionalPipeService` 's constructor takes a `PeerGroup` object as its sole argument, which it uses to extract the peer group's Discovery and Pipe service objects:

```
public BidirectionalPipeService (PeerGroup peerGroup);
```

As shown in Figure 8.9, `BidirectionalPipeService` provides only two other methods: `bind` and `connect`. The `bind` method is used to create an instance of `AcceptPipe`, an inner class defined by `BidirectionalPipeService`, which uses an input pipe to listen for connections from other peers. The `connect` method is used to connect to a remote peer that is already listening for connections. `BidirectionalPipeService` and its support classes use a clever trick to require you to work directly with only one Pipe Advertisement.

Figure 8.9. Flow of `BidirectionalPipeService` messages.



When the `connect` method is called, the Pipe Advertisement passed to the method binds an output pipe. If that output pipe is bound successfully, the `connect` method creates and binds a new input pipe. The `connect` method sends this new pipe's advertisement to the remote peer using the newly bound output pipe. On the remote peer, the `AcceptPipe` object listening for new connections receives the Pipe Advertisement and uses it to bind an output pipe. The remote peer can now use this output pipe to send messages back to the originating peer. The remote peer creates one more Pipe Advertisement and binds an input pipe using this advertisement. This advertisement is sent as an acknowledgement, which means that the original pipe used to negotiate the two-way communications channel is no longer used. The peer receiving the acknowledgement advertisement uses it rather than the original pipe to send messages to the remote peer.Voilá—two-way communication. Using `BidirectionalPipeService`, you can combine the earlier examples in this chapter to create the simple chat client in Listing 8.8.

Listing 8.8 Source Code for *PipeClientServer.java*

```
package com.newriders.jxta.chapter8;

import java.awt.BorderLayout;
import java.awt.FlowLayout;
```

```
import java.awt.Container;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import java.io.FileWriter;
import java.io.FileInputStream;
import java.io.FileNotFoundException;
import java.io.IOException;

import java.net.URL;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredTextDocument;

import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;

import net.jxta.impl.util.BidirectionalPipeService;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;

import net.jxta.pipe.InputPipe;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.PipeService;
```

```
import net.jxta.protocol.PipeAdvertisement;


/**
 * A simple text messaging application, which uses a bidirectional
 * pipe to send and receive messages.
 */
public class PipeClientServer
    implements BidirectionalPipeService.MessageListener
{
    /**
     * The peerGroup for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * The frame for the user interface.
     */
    private JFrame clientFrame = new JFrame("PipeClientServer");

    /**
     * The text field in the user interface to accept the message
     * text to be sent over the pipe.
     */
    private JTextField messageText = new JTextField(20);

    /**
     * A label used to display the received message in the GUI.
     */
    private JLabel receivedText = new JLabel(
        "Waiting to receive a message...");

    /**
     * Indicates whether the pipe has been bound already.
     */
    private boolean initialized = false;
```

```
/**
 * The bidirectional pipe object to use to send and receive messages.
 */
private BidirectionalPipeService.Pipe pipe = null;


/**
 * Creates an input pipe and its advertisement using the
 * BidirectionalPipeService. This is used when starting this class up
 * in "server" mode. The advertisement is saved to file so that another
 * instance of this class can use the advertisement to start up in
 * "client" mode.
 */
public void createPipeAdv() throws IOException
{
    BidirectionalPipeService pipeService = new
        BidirectionalPipeService(peerGroup);

    // Create an accept pipe to use to create an input pipe and
    // listen for connections. "PipeClientServer" is simply the
    // symbolic name that will appear in the Pipe Advertisement
    // created by the BidirectionalPipeService.
    BidirectionalPipeService.AcceptPipe acceptPipe =
        pipeService.bind("PipeClientServer");

    // Extract the Pipe Advertisement and write it to file.
    PipeAdvertisement pipeAdv = acceptPipe.getAdvertisement();
    try
    {
        FileWriter file = new FileWriter("PipeClientServer.xml");
        MimeMediaType mimeType = new MimeMediaType("text/xml");
        StructuredTextDocument document =
            (StructuredTextDocument) pipeAdv.getDocument(mimeType);

        // Output the XML for the advertisement to the file.
```

```
            document.sendToWriter(file);
            file.close();
        }
        catch (Exception e)
        {
            e.printStackTrace();
        }

        // "Accept" a connection, meaning set up the input pipe and listen
        // for messages. Set this object as the MessageListener so that
        // we can handle incoming messages without having to spawn a
        // thread to call waitForMessage on the input pipe.
        while (null == pipe)
        {
            try
            {
                pipe = acceptPipe.accept(30000, this);
            }
            catch (InterruptedException e)
            {
                System.out.println("Error trying to accept(): " + e);
            }
        }

        // Show the user interface.
        showGUI();
    }

    /**
     * Starts the JXTA platform.
     *
     * @exception   PeerGroupException thrown if the platform can't
     *               be started.
     */
    public void initializeJXTA() throws PeerGroupException
    {
```

```
    peerGroup = PeerGroupFactory.newNetPeerGroup();
}

/**
 * Starts the class in "client" mode, loading a Pipe Advertisement from
 * the given file. This advertisement is used to create an output pipe
 * to talk to the remote peer and set up the bidirectional
 * communications channel.
 *
 * @param      fileName the name of the file from which to load the
 *             Pipe Advertisement.
 * @exception  FileNoteFoundException if the Pipe Advertisement
 *             file can't be found.
 * @exception  IOException if there is an error binding the pipe.
 */
public void loadPipeAdv(String fileName)
    throws FileNotFoundException, IOException
{
    FileInputStream file = new FileInputStream(fileName);
    MimeMediaType asMimeType = new MimeMediaType("text/xml");

    // Load the advertisement.
    PipeAdvertisement pipeAdv =
        (PipeAdvertisement) AdvertisementFactory.newAdvertisement(
            asMimeType, file);

    // Connect using the Pipe Advertisement and the
    // BidirectionalPipeService.
    BidirectionalPipeService pipeService =
        new BidirectionalPipeService(peerGroup);

    while (null == pipe)
    {
        try
        {
            System.out.println("Trying...");
```

```
            pipe = pipeService.connect(pipeAdv, 30000);
            System.out.println("Done Trying...");
        }
        catch (IOException e)
        {
            // Do nothing.
        }
    }

    // Show the user interface.
    showGUI();

    // There is no way to register a listener with the input pipe used
    // by the Pipe object to receive message. So, use the
    // waitForMessage method instead. Not the nicest way to do this,
    // but it gives you the idea.
    InputPipe input = pipe.getInputPipe();

    while (true)
    {
        try
        {
            Message message = input.waitForMessage();

            // Set the user interface to display the message text.
            receivedText.setText(message.getString("MessageText"));
        }
        catch (InterruptedException e)
        {
            // Do nothing, ignore the interruption.
        }
    }
}

/**
 * Runs the application. The application can run in either "server" or
```

```
     * "client" mode. In "server" mode, the application creates a new Pipe
     * Advertisement, writes it to a file, and binds an input pipe to start
     * listening for incoming messages. In "client" mode, a Pipe
     * Advertisement is read from a file and used to bind an output pipe
to
     * a remote peer.
     *
     * @param   args the command-line arguments passed to the application.
     */
    public static void main(String[] args)
    {
        PipeClientServer client = new PipeClientServer();

        if (args.length == 0)
        {
            // No arguments, therefore we must be trying to
            // set up a new server. Create a input pipe and
            // write its advertisement to a file.
            try
            {
                // Initialize the JXTA platform.
                client.initializeJXTA();

                // Create the input connection and save the
                // Pipe Advertisement.
                client.createPipeAdv();
            }
            catch (PeerGroupException e)
            {
                System.out.println("Error starting JXTA platform: " + e);
                System.exit(1);
            }
            catch (FileNotFoundException e2)
            {
                System.out.println("Unable to load Pipe Advertisement: "
                    + e2);
```

```
            System.exit(1);
        }
        catch (IOException e3)
        {
            System.out.println("Error loading or binding Pipe"
                + " Advertisement: " + e3);
            System.exit(1);
        }
    }
    else if (args.length == 1)
    {
        // If there's one argument, then we need to try to
        // connect to an existing server using the Pipe Advertisement
        // in the file specified by the argument.
        try
        {
            // Initialize the JXTA platform.
            client.initializeJXTA();

            // Load the Pipe Advertisement.
            client.loadPipeAdv(args[0]);
        }
        catch (PeerGroupException e)
        {
            System.out.println("Error starting JXTA platform: " + e);
            System.exit(1);
        }
        catch (FileNotFoundException e2)
        {
            System.out.println("Unable to load Pipe Advertisement: "
                + e2);
            System.exit(1);
        }
        catch (IOException e3)
        {
            System.out.println("Error loading or binding Pipe"
```

```
                + " Advertisement: " + e3);
            System.exit(1);
        }
    }
    else
    {
        System.out.println("Usage:");
        System.out.println("'server' mode: PipeClientServer");
        System.out.println("'client' mode: PipeClientServer "
            + "<filename>");
    }
}

/**
 * Handles displaying an incoming message to the user interface.
 *
 * @param   message the message received by the input pipe.
 * @param   pipe an OutputPipe to use to send a response.
 */
public void messageReceived(Message message, OutputPipe pipe)
{
    // Set the user interface to display the message text.
    receivedText.setText(message.getString("MessageText"));
}

/**
 * Sends a message string to the remote peer using the output pipe.
 *
 * @param   messageString the message text to send to the remote peer.
 */
private void sendMessage(String messageString)
{
    PipeService pipeService = peerGroup.getPipeService();
    OutputPipe outputPipe = null;

    // Create and configure a message object.
```

```
        Message message = pipeService.createMessage();
        message.setString("MessageText", messageString);

        // Get the output pipe from the pipe.
        outputPipe = pipe.getOutputPipe();
        if (null != outputPipe)
        {
            try
            {
                // Send the message.
                outputPipe.send(message);
            }
            catch (IOException e)
            {
                // Show some warning dialog.
                JOptionPane.showMessageDialog(null, e.toString(),
"Error",
                    JOptionPane.WARNING_MESSAGE);
            }
        }
        else
        {
            // Show some warning dialog.
            JOptionPane.showMessageDialog(null, "Output pipe is null!",
                "Error", JOptionPane.WARNING_MESSAGE);
        }
    }

    /**
     * Configures and displays a simple user interface to allow the user
to
     * send text messages. The GUI also allows the user to stop the client
     * application.
     */
    private void showGUI()
    {
```

```
JButton sendButton = new JButton("Send Message");
JButton quitButton = new JButton("Quit");

JPanel receivePane = new JPanel();
receivePane.setLayout(new FlowLayout());
receivePane.add(receivedText);

JPanel sendPane = new JPanel();
sendPane.setLayout(new FlowLayout());
sendPane.add(messageText);
sendPane.add(sendButton);
sendPane.add(quitButton);

// Populate the GUI frame.
Container pane = clientFrame.getContentPane();
pane.setLayout(new BorderLayout());
pane.add(receivePane, BorderLayout.NORTH);
pane.add(sendPane, BorderLayout.SOUTH);

// Set up listeners for the buttons.
sendButton.addActionListener(
    new ActionListener() {
        public void actionPerformed(ActionEvent e)
        {
            // Send the message.
            sendMessage(messageText.getText());

            // Clear the text.
            messageText.setText("");
        }
    }
);
quitButton.addActionListener(
    new ActionListener()
    {
        public void actionPerformed(ActionEvent e)
```

```
                {
                    clientFrame.hide();

                    // Stop the JXTA platform. Currently, there isn't any
                    // nice way to do this.
                    System.exit(0);
                }
            }
        );

        // Pack and display the user interface.
        clientFrame.pack();
        clientFrame.show();
    }
}
```

The `PipeClientServer` example has two modes of operation:

- **Server mode—** This mode is used when you want to start a new bidirectional pipe. It causes a new Pipe Advertisement to be written to the file `PipeClientServer.xml`. This advertisement is used by any peer that wants to send messages to the peer and initiate a bidirectional connection.

- **Client mode—** This mode is used when you want to connect to an existing bidirectional pipe. To connect, you need to provide a Pipe Advertisement as part of the command-line arguments. In this example, the advertisement that must be provided is the `PipeClientServer.xml` file written by another instance of `PipeClientServer`, running in server mode.

To see this example in operation, you need to use two separate instances of `PipeClientServer`. This requires two separate directories containing the compiled source code and JXTA JARs. As in previous examples, you need to configure the JXTA platform for each directory to use different TCP and HTTP ports.

After you create and configure the two directories, run one instance of `PipeClientServer` in server mode from one directory using this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar com.newriders.jxta.chapter8.PipeClientServer
```

After the bidirectional pipe has successfully created and published a Pipe Advertisement, you should see a message on the console similar to this:

```
Published bidir pipe
urn:jxta:uuid-59616261646162614E50472050325033D7F4C6E7
B2BD4572B6628F1DFEE6B34404
```

This indicates that the bidirectional pipe has created an input pipe and published the pipe's advertisement. The `PipeClientServer` extracts this advertisement and writes it to the file `PipeClientServer.xml`.

Now that an input pipe has been started, you need to start a second instance of `PipeClientServer`, this time in client mode. To do this, you need to provide a Pipe Advertisement. Copy the `PipeClientServer.xml` file from the first `PipeClientServer` instance's directory to the directory where the second instance of `PipeClientServer` will be started. Start the second instance of `PipeClientServer` from this directory using this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar com.newriders.jxta.chapter8.PipeClientServer
PipeClientServer.xml
```

The second instance loads the Pipe Advertisement from `PipeClientServer.xml` and attempts to bind an output pipe. After this has been done, both instances of `PipeClientServer` should display their user interfaces, as shown in Figure 8.10.

Figure 8.10. The `PipeClientServer` user interface.

You should now be able to send messages between the two `PipeClientServer` instances. Note that if you stop both instances and start them again, you need to recopy the `PipeClientServer.xml` file created by the server mode instance. The bidirectional pipe creates a new Pipe Advertisement each time.

Summary

In this chapter, you learned how a pipe can be bound to an endpoint to send data to or receive data from a remote peer. To demonstrate the use of the Pipe service, this chapter used a set of Pipe Advertisement files generated by the `PipeAdvPopulator` class. Although these files simplified the examples, it should be realized that in real applications, Pipe Advertisements usually are obtained using the Discovery service.

This chapter also examined the `BidirectionalPipeService`, a pseudo-service built on top of the Pipe service. The `BidirectionalPipeService` provides a simple mechanism for peers to establish two-way communications using two pipes. The advantage of this mechanism is that only one Pipe Advertisement must be published or discovered because the `BidirectionalPipeService` handles negotia-tion of a second Pipe Advertisement. This mechanism also has the advantage that it eliminates some of the code required to manage two pipes.

Pipe Advertisements aren't usually published by themselves, but they are usually contained within another advertisement. As you'll see in Chapter 10, "Peer Groups and Services," a Pipe Advertisement is usually associated with a service, allowing a remote peer to interact with a service through the pipe.

Chapter 9. The Endpoint Routing Protocol

Due to the ad hoc nature of a P2P network, a message between two endpoints might need to travel through intermediaries. An intermediary might be used to allowing peers with incompatible network transports to communicate by using the intermediary as a gateway. To determine how a message should be sent between two endpoints, a mechanism is required to allow a peer to discover route information. The Endpoint Routing Protocol (ERP) provides peers with a mechanism for determining a route to an endpoint, allowing the peer to send data to the remote endpoint.

Before learning about the Endpoint Routing Protocol, it is necessary to understand how endpoints work. In Chapter 8, "The Pipe Binding Protocol," you learned that although pipes provide a transmission mechanism, the pipes themselves are not responsible for the actual transmission and reception of data. Pipes are an abstraction built on top of endpoints to provide a convenient programming model. Endpoints are the entity responsible for conducting the actual exchange of information over a network. Endpoints encapsulate a set of network interfaces, allowing a peer to send and receive data independently of the type of network transport being employed.

Although JXTA provides the Resolver and Pipe services to enable highlevel use of endpoints, some services might want to use endpoints directly. This chapter first explores the use of endpoints for conducting network communication and then details the Endpoint Routing Protocol and its relationship to endpoints.

Introduction to Endpoints

JXTA offers two simple ways to send and receive messages: the Resolver service and the Pipe service. However, as revealed in Chapter 5, "The Peer Resolver Protocol," and

Chapter 8, these services are simply convenient wrappers for sending and receiving messages using a peer's local endpoints. An *endpoint* is an interface to a set of network transports that allows data to be sent across the network. In JXTA, network transports are assumed to be unreliable, even though actual endpoint protocol implementations might use reliable transports such as TCP/IP.

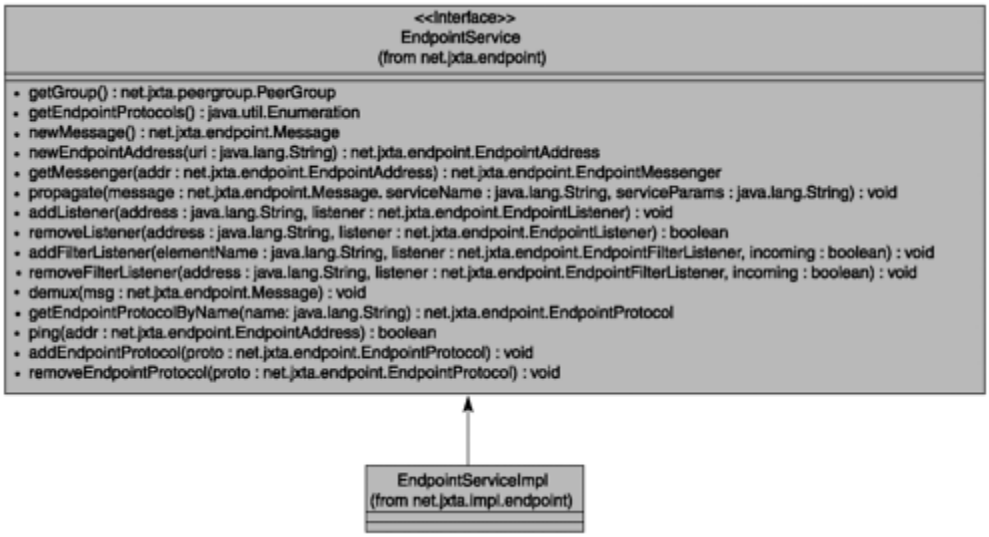
Unlike other areas of the JXTA platform, endpoint functionality doesn't have a protocol definition. Details on how data is to be formatted for transport across the network is the responsibility of a particular endpoint protocol implementation. The only functionality exposed to the developer is provided by the Endpoint service, which aggregates the registered endpoint protocol implementations for use by a developer. Although a developer could use the Endpoint service implementation to obtain a particular endpoint protocol implementation and use it directly, this is not desirable in most cases. Using a particular endpoint protocol implementation directly makes a solution less flexible by making the solution dependent on a particular network transport.

The Endpoint Service

The Endpoint service provides an access point to all the endpoint protocol implementations installed on a peer, allowing a programmer to send a message using these endpoint protocol implementations. Unlike the other core services in JXTA, the Endpoint service is independent of a peer group. All peer groups share the same Endpoint service, which makes sense, considering that the Endpoint service provides the communication layer closest to the network transport layer. By default, a peer group in the reference implementation inherits the Endpoint service provided by its parent group. However, developers can provide a custom Endpoint service implementation for a peer group that they create by loading a custom Endpoint service. This Endpoint service implementation is loaded just like any other custom service, using the techniques demonstrated in Chapter 10, "Peer Groups and Services."

In the reference implementation, the Endpoint service, shown in Figure 9.1, is defined by the `EndpointService` interface in the `net.jxta.endpoint` package and is implemented by the `EndpointServiceImpl` class in the `net.jxta.impl.endpoint` package.

Figure 9.1. The Endpoint service interface and implementation.



Although the endpoint protocol implementations define the format for data crossing the network, the Endpoint service does add one piece of information when propagating a message. The Endpoint service adds a message element named `jxta:EndpointHeaderSrcPeer` to the outgoing messages. If visualized as XML, remembering that messages aren't necessarily rendered to XML, the message element's format would be as follows:

```
<jxta:EndpointHeaderSrcPeer>
  . . .
</jxta:EndpointHeaderSrcPeer>
```

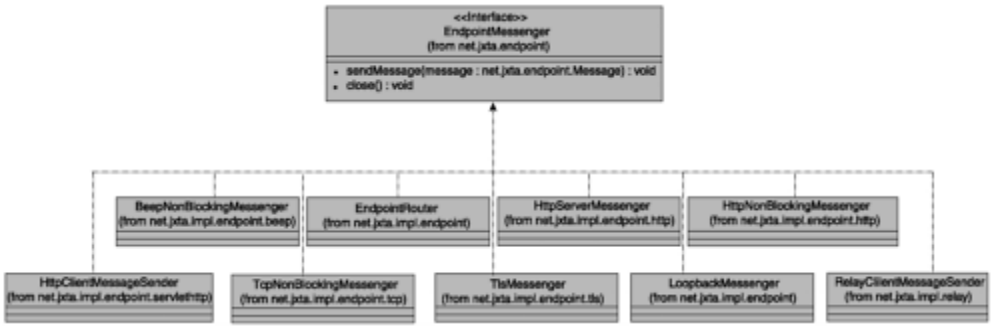
The `jxta:EndpointHeaderSrcPeer` element contains the ID of the peer propagating the message. This Peer ID is used by the Endpoint service that receives the message to eliminate loopback by discarding messages whose source Peer ID matches the local Peer ID. The remainder of the formatting of an outgoing message is the responsibility of a particular endpoint protocol implementation registered with an `EndpointService` instance using the `EndpointService.addEndpointProtocol` method.

As shown in Figure 9.2, an endpoint protocol implementation realizes the `EndpointProtocol` interface from the `net.jxta.endpoint` package.

Figure 9.2. The `EndpointProtocol` interface and implementations.

An `EndpointProtocol` implementation allows a peer to propagate a message to as many peers as possible. In a TCP endpoint protocol implementation, for example, TCP multicast capabilities are used to send a message to as many peers on the local LAN segment as possible. An `EndpointProtocol` implementation is also responsible for allowing a peer to send a message directly to a peer located at a specific Endpoint Address. This functionality is provided by an implementation of the `EndpointMessenger` interface obtained using the `EndpointProtocol` implementation's `getMessenger` method. The `EndpointMessenger` interface, shown in Figure 9.3, is defined in `net.jxta.endpoint`.

Figure 9.3. The `EndpointMessenger` interface and implementations.



An implementation of `EndpointMessenger` is usually obtained using the `EndpointService.getMessenger` method. The `EndpointService.getMessenger` method takes a `net.jxta.endpoint.EndpointAddress` argument that identifies the remote peer's transport-specific location. This address is used to determine which `EndpointProtocol` provides connectivity to the remote peer, and it calls `getMessenger` on the `EndpointProtocol` implementation. This encapsulation and abstraction eliminate the need for a developer to ever instantiate an `EndpointMessenger` implementation directly.

Types of Endpoint Transport Implementations

In the Java reference implementation, currently five endpoint protocol implementations are available:

- **TCP (`net.jxta.impl.endpoint.tcp`)—** This provides a TCP `EndpointProtocol` implementation that uses a `MulticastSocket` to send data to peers on the local LAN segment. A TCP-based `EndpointMessenger` implementation uses a `Socket` to connect directly to a remote peer.

- **HTTP (`net.jxta.impl.endpoint.http`)—** This provides an HTTP `EndpointProtocol` and `EndpointMessenger`. This endpoint protocol is slightly different from a typical endpoint protocol implementation because the HTTP endpoint protocol implementation provides the router peer functionality that allows peers to perform firewall traversal. The HTTP implementation of `EndpointProtocol` does not provide broadcast capabilities.

- **Servlet HTTP (`net.jxta.impl.endpoint.servlethttp`)—** Similar to the HTTP implementation, the Servlet HTTP implementation provides HTTP transport functionality that can be plugged into application servers that support the Java Servlet APIs.

- **TLS (`net.jxta.impl.endpoint.tls`)—** This is the Transport Layer Security protocol endpoint protocol implementation. This endpoint protocol implementation does not provide broadcast capabilities because the TLS implementation is designed only for securing one-to-one communications. This implementation is built on top of libraries provided by the Cryptix project (www.cryptix.org).

- **BEEP (`net.jxta.impl.endpoint.beep`)—** This is the Block Extensible Exchange Protocol (IETF RFC 3080) implementation. BEEP is basically a framework for building application protocols. This endpoint protocol implementation does not provide broadcast capabilities. This implementation is built on top of libraries provided by `beepcore.org`.

One other implementation, the Endpoint Router implementation, provides a transport that handles finding routes to remote peers via gateways. This transport provides the implementation of the Endpoint Routing Protocol that will be discussed later in this chapter.

Each endpoint protocol implementation made available by a peer is identified by a Transport Advertisement in the peer's Peer Advertisement. However, the format of this Transport Advertisement varies by endpoint protocol implementation. Only the root element is common to all implementations:

```
<jxta:TransportAdvertisement>
  . . .
</jxta:TransportAdvertisement>
```

By default, the endpoint protocol implementations are configured when the JXTA platform boots. A default set of endpoint protocol implementations is added to the Endpoint service based on the settings provided by the user to the JXTA configuration tool. Currently, the default transports loaded include TCP, HTTP, and TLS.

Endpoint Addresses

Endpoint Addresses provide the network transport-specific information required to route a message over a particular endpoint protocol implementation to a specific peer and service. In general, the format of an Endpoint Address in the reference implementation takes this form:

```
<protocol>://<network address>/<service name>/<service parameters>
```

The following definitions are used for each section of the Endpoint Address:

- **`<protocol>`**— The name of the network transport to use when sending the message. Example values include `tcp`, `http`, and `jxtatls`.

- **<protocolAddress>—** The network transport-specific address used to locate the destination peer on the network. For example, a TCP Endpoint Address would use an IP address and port number for this value.

- **<serviceName>—** An identifier that uniquely specifies the destination service on the remote peer. This effectively allows messages arriving over a single network transport to be demultiplexed by the Endpoint service and passed to the appropriate service. To associate a service with a particular peer group, the service name is usually a combination of a common name for the service and the Peer Group ID.

- **<serviceParameters>—** Some unique identifying parameters being passed to the service. These parameters might be used by a particular destination service to provide information required to route the message to a particular handler instance before parsing the message itself.

For example, a message destined for the Pipe service on a remote peer using the TCP endpoint protocol implementation would use an Endpoint Address that looks like this:

```
tcp://10.6.18.38:80/PipeService/<Pipe ID>
```

In this example, Endpoint Address, `10.6.18.38:80` is the destination's IP address (10.6.18.38) and port number (80), `PipeService` is the name of the service, and `<Pipe ID>` is the parameter to the Pipe service.

Only the protocol and the network address are required elements in the Endpoint Address. If the Endpoint Address has no service specified, the form of the address changes slightly to this:

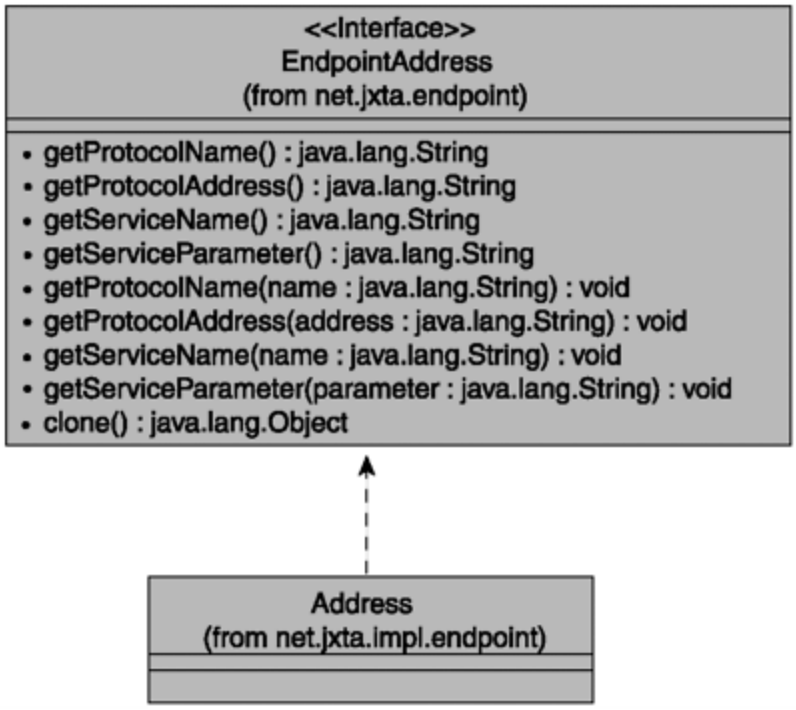
```
<protocol>://<network address>/
```

Note that, in this case, no service parameters are specified because no service has been specified. An address may also specify a service name but no service parameters, in which case the form of the address is as follows:

```
<protocol>://<network address>/<service name>
```

The reference implementation defines the `EndpointAddress` interface in `net.jxta.endpoint` and the `Address` implementation in `net.jxta.impl.endpoint`, shown in Figure 9.4, to handle the details of manipulating Endpoint Addresses.

Figure 9.4. The `EndpointAddress` interface and implementation.



Instead of specifying an Endpoint Address in a transport-specific form, such as `tcp://10.6.18.38`, higher-level services in JXTA use a transport-neutral Endpoint Address of this form:

```
jxta://<unique Peer ID>
```

The `jxta` protocol specifier is used to indicate the JXTA-specific Endpoint Routing Protocol. This form of Endpoint Address is used to allow JXTA peers to act independently of the network transport. By using the `jxta` form of the address, a peer can send messages via the Endpoint Routing Protocol as if connecting directly to the remote peer. In fact, the message might travel through several peers, a fact that is unknown to the peer. In this way, the Endpoint Routing Protocol abstracts the details of the underlying network topology, allowing a peer to act as if it is capable of connecting directly to a remote peer.

Message Formatting

Unlike the other services in JXTA, no corresponding protocol defines the format of messages sent by the Endpoint service. Although the endpoint protocol implementations are ultimately responsible for handling the details of formatting a message, the reference protocol implementations share code to render a message from the internal XML object structure into a format suitable for transport over the network. Currently, a transport can use two possible output formats to render a `Message` object:

- **Binary message format—** The message elements are rendered into simple binary byte stream. This functionality is encapsulated in the `MessageWireFormatBinary` class in the `net.jxta.impl.endpoint` package. This format of the output produced by this class is specified by the `application/x-jxta-msg` MIME type.

- **XML message format—** The message is rendered from the `Message` object's representation of an XML tree into real XML output. This functionality is encapsulated in the `MessageWireFormatXML` class in the `net.jxta.impl.endpoint` package. This format of the output produced by this class is specified by the `text/xml` MIME type.

Both `MessageWireFormatXML` and `MessageWireFormatBinary` extend the `MessageWireFormat` abstract class. Endpoint protocol implementations create an instance of a specific type of wire-formatting object using the `MessageWireFormatFactory` class and specifying the appropriate MIME type to the `newMessageWireFormat` method. It is up to the endpoint protocol implementation to choose the output format most appropriate to its particular network transport.

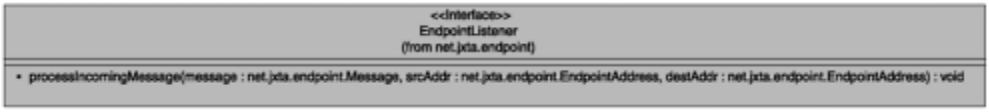
Using the Endpoint Service

To demonstrate the use of the Endpoint service, this section develops an application similar to the one in Chapter 7, "The Peer Information Protocol." The difference is that this example uses the Endpoint service on which pipes are built to provide the messaging functionality.

Receiving Incoming Messages

It should come as no surprise that the `EndpointService` is structured in a similar way to those services built on top of it. The Endpoint service provides the `EndpointListener` interface in `net.jxta.endpoint`, shown in Figure 9.5, to allow other services to receive notification of arriving messages.

Figure 9.5. The `EndpointListener` interface.



The sole method that developers need to implement, `processIncomingMessage`, accepts the arriving `Message` object as well as the source and destination Endpoint Addresses:

```
public void processIncomingMessage(Message message,
    EndpointAddress source, EndpointAddress destination);
```

To listen for messages arriving for a specific service, a developer needs only to register an `EndpointListener` instance with the `EndpointService` instance using the `EndpointService.addListener` method:

```
public void addListener(String address, EndpointListener listener)
        throws IllegalArgumentException;
```

The `EndpointListener` 's `processIncomingMessage` method is called whenever a `Message` arriving at the peer contains a destination Endpoint Address that specifies a service matching the `address` value.

Note that when registering an `EndpointListener`, the `address` value that you pass shouldn't be just the name of the destination service. Instead, the `address` should be the concatenation of the destination service name and service parameters that are part of the

destination Endpoint Address. This is an area that is ambiguous in the current reference implementation and will be refined in future releases.

To demonstrate the use of the `EndpointListener` and `EndpointService` interfaces, the `EndpointServer` example in Listing 9.1 starts the JXTA platform and adds itself to the `EndpointService` instance as a listener for messages addressed to a service with the name `EndpointServer` plus the same Peer Group ID with the parameters `012345`.

Listing 9.1 Source Code for *EndpointServer.java*

```
package com.newriders.jxta.chapter9;

import java.util.Enumeration;

import net.jxta.endpoint.EndpointAddress;
import net.jxta.endpoint.EndpointProtocol;
import net.jxta.endpoint.EndpointService;
import net.jxta.endpoint.EndpointListener;
import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;

import net.jxta.impl.endpoint.Address;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;



/**
 * A simple server that listens on an endpoint, looking for
 * Messages destined for a service named EndpointServer
 * concatenated with the Peer Group ID, with service
 * parameters 012345.
 */
public class EndpointServer implements EndpointListener
{
```

```
/**
 * The peer group for the application.
 */
private PeerGroup peerGroup = null;

/**
 * The service name to use when listening for messages.
 * This service name will be appended with the Peer Group ID
 * of the peer group when the JXTA platform is started.
 */
private String serviceName = "EndpointServer";

/**
 * The service parameters to use when listening for
 * messages.
 */
private String serviceParameters = "012345";


/**
 * Starts the JXTA platform.
 *
 * @exception   PeerGroupException thrown if the platform
 *              can't be started.
 */
public void initializeJXTA() throws PeerGroupException
{
    peerGroup = PeerGroupFactory.newNetPeerGroup();

    // Add the Peer Group ID of the group to the service
    // name so the endpoint listener is specific to
    // the peer group.
    serviceName += peerGroup.getPeerGroupID().toString();
}

/**
```

```
     * Runs the application: starts the JXTA platform, starts
     * listening on the Endpoint service for messages.
     *
     * @param   args the command-line arguments passed to
     *     the application.
     */
    public static void main(String[] args)
    {
        EndpointServer server = new EndpointServer();

        try
        {
            // Initialize the JXTA platform.
            server.initializeJXTA();

            // Start the server.
            server.start();
        }
        catch (PeerGroupException e)
        {
            System.out.println("Error starting JXTA platform: "
                + e);
            System.exit(1);
        }
    }

    /**
     * The EndpointListener implementation. Accepts an incoming
     * message for processing.
     *
     * @param   message the Message that has arrived for
     *          processing.
     * @param   source the EndpointAddress of the peer sending
     *          the message.
     * @param   destination the EndpointAddress of the
     *          destination peer for the message.
```

```
 */
public void processIncomingMessage(Message message,
    EndpointAddress source, EndpointAddress destination)
{
    System.out.println("Message received from " + source
        + " for " + destination + ":");
    System.out.println(message.getString("MessageText"));
}

/**
 * Start the server listening on the Endpoint service.
 */
public void start()
{
    EndpointService endpoint =
        peerGroup.getEndpointService();

    // Print out all of the endpoint protocol addresses.
    // These can be used by the EndpointClient to send a
    // message to the EndpointServer.
    EndpointProtocol aProtocol = null;
    Enumeration protocols = endpoint.getEndpointProtocols();
    while (protocols.hasMoreElements())
    {
        aProtocol =
            (EndpointProtocol) protocols.nextElement();

        // Print out the address.
        System.out.println("Endpoint address: "
            + aProtocol.getPublicAddress().toString());
    }

    // Add ourselves as a listener to the Endpoint service.
    endpoint.addListener(serviceName + serviceParameters,
         this);
}
```

```
}
```

By itself, the `EndpointServer` example isn't very useful without another peer capable of sending messages to the EndpointServer service for the peer group. Peers can propagate a message to many peers using the Endpoint service or send a message directly to a specific peer using an `EndpointMessenger`.

Propagating Messages Using the Endpoint Service

Propagating a message to a number of remote peers works in a similar fashion to using propagation pipes, but without the requirement for you to find and bind an output pipe. However, unlike propagation pipes, the Endpoint service cannot propagate messages across firewall and NAT boundaries. Propagation across firewalls and network boundaries is a feature offered by the Rendezvous service, explained in Chapter 6, "The Rendezvous Protocol," which builds on the Endpoint service to provide this capability. Propagation using the Endpoint service is built on the capabilities of registered endpoint protocol implementations to broadcast to a number of Endpoint Addresses simultaneously. This functionality is not available in all network transports, such as HTTP, but it is available in low-level network transports, such as TCP. The reference implementation of the Endpoint service provides propagation using only the TCP endpoint protocol implementation and is thus limited to propagating messages within the boundaries of a LAN segment.

`EndpointPropagateClient` in Listing 9.2 provides a simple example of the elements necessary to propagate a message using the `EndpointService` instance.

Listing 9.2 Source Code for *EndpointPropagateClient.java*

```
package com.newriders.jxta.chapter9;

import java.io.IOException;

import net.jxta.endpoint.EndpointAddress;
import net.jxta.endpoint.EndpointService;
import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;
```

```
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;


/**
 * A simple client that uses the Endpoint service to propagate
 * messages to a service named EndpointServer concatenated
 * with the Peer Group ID, with the service parameters 012345
 * on all peers in the local LAN segment.
 */
public class EndpointPropagateClient
{
    /**
     * The peer group for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * The service name to use when listening for messages.
     * This service name will be appended with the Peer Group ID
     * of the peer group when the JXTA platform is started.
     */
    private String serviceName = "EndpointServer";

    /**
     * The service parameters to use when listening for
     * messages.
     */
    private String serviceParameters = "012345";
    /**
     * Starts the JXTA platform.
     *
     * @exception   PeerGroupException thrown if the platform
     *              can't be started.
     */
```

```
public void initializeJXTA() throws PeerGroupException
{
    peerGroup = PeerGroupFactory.newNetPeerGroup();

    // Add the Peer Group ID of the group to the service
    // name so the message is sent to the version of the
    // endpoint listener specific to this peer group.
    serviceName += peerGroup.getPeerGroupID().toString();
}

/**
 * Runs the application: starts the JXTA platform, accepts
 * user input messages, and propagates them to other peers.
 *
 * @param   args the command-line arguments passed to the
 *     application.
 */
public static void main(String[] args)
{
    EndpointPropagateClient client =
        new EndpointPropagateClient();

    try
    {
        boolean done = false;
        String messageString = null;

        // Initialize the JXTA platform.
        client.initializeJXTA();

        while (!done)
        {
            // Reset the message string.
            messageString = null;

            // Get the message; if the message is '.',
```

```
            // then quit the application.
            System.out.print("Enter a message (or '.' "
                + "to quit): ");
            messageString = client.readInput();

            if ((messageString.length() > 0)
                && (!messageString.equals(".")))
            {
                // Send a message to the server.
                client.sendMessage(messageString);
            }
            else
            {
                // We're done.
                done = true;
            }
        }

        // Stop the JXTA platform. Currently, there isn't
        // any nice way to do this.
        System.exit(0);
    }
    catch (PeerGroupException e)
    {
        System.out.println("Error starting JXTA platform: "
            + e);
        System.exit(1);
    }
}

/**
 * Read a line of input from the system console.
 *
 * @return  the String read from the System.in InputStream.
 */
public String readInput()
```

```
{
    StringBuffer result = new StringBuffer();
    boolean done = false;
    int character;
    while (!done)
    {
        try
        {
            // Read a character.
            character = System.in.read();

            // Check to see if the character is a newline.
            if ((character == -1)
                || ((char) character == '\n'))
            {
                done = true;
            }
            else
            {
                // Add the character to the result string.
                result.append((char) character);
            }
        }
        catch (IOException e )
        {
            done = true;
        }
    }

    return result.toString().trim();
}

/**
 * Sends a message. In this case, the message string is
 * propagated to all peers in the peer group on the local
 * LAN segment.
```

```
     *
     * @param   messageString the message to send to other
     *          peers.
     */
    public void sendMessage(String messageString)
    {
        EndpointService endpoint =
            peerGroup.getEndpointService();

        // Create a new message.
        Message message = endpoint.newMessage();

        // Populate the message contents with the messageString.
        message.setString("MessageText", messageString);

        try
        {
            // Propagate the message within the peer group.
            endpoint.propagate(message, serviceName,
                serviceParameters);
        }
        catch (IOException e)
        {
            System.out.println("Error sending message: " + e);
        }
    }
}
```

To propagate a message, create a `Message` object using the `EndpointService` 's `createMessage` method, and populate it in the same fashion as when sending a message using a pipe. As with any `Message`, multiple elements can be added. In the example, a single element called `MessageText` containing the outgoing text being sent to the remote peer is added using the following code:

```
// Populate the message contents with the messageString.
```

```
message.setString("MessageText", messageString);
```

It is propagated to other peers using this code:

```
// Propagate the message within the peer group.
endpoint.propagate(message, serviceName, serviceParameters);
```

The `EndpointService.propagate` method takes not only the message being propagated, but also the name of the destination service and parameters to pass to the destination service.

Using EndpointServer and EndpointPropagateClient

As with the `PipeServer` and `PipeClient` examples created in Chapter 8, using `EndpointServer` and `EndpointPropagateClient` requires two separate instances of the JXTA platform. To prepare to run the `EndpointServer` and `EndpointPropagateClient` examples, follow these steps:

1. Create two directories, placing the `EndpointServer` source code in one directory and the `EndpointPropagateClient` source code in the other.

2. Copy all of the JAR files from the `lib` directory under the JXTA Demo `install` directory into each directory.

3. Start a command console and change to the directory containing the `EndpointServer` code.

4. Compile `EndpointServer` using `javac -d . -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;log4j.jar;minimalBC.jar EndpointServer.java.`

5. Start the EndpointServer using `java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.`

```
jar;log4j.jar;minimalBC.jar
com.newriders.jxta.chapter9.EndpointServer.
```

`EndpointServer` starts and prints the Endpoint Address for each of the protocols registered with the `EndpointService` instance. This isn't used in this example, but it will be used when demonstrating the use of `EndpointMessenger`.

6. Start a second command console and change to the directory containing the `EndpointPropagateClient` code.

7. Compile `EndpointPropagateClient` using `javac -d . -classpath .;beepcore .jar;cms.jar;cryptix32.jar;cryptix-asn1.jar; instantp2p.jar;jxta.jar; jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;log4j.jar;minimalBC.jar EndpointPropagateClient.java.`

8. Start `EndpointPropagateClient` using `java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;instantp2p.jar;jxta.jar;jxtaptls. jar;jxtasecurity.jar;jxtashell.jar;log4j.jar;minimalBC.jar com.newriders.jxta.chapter9.EndpointPropagateClient.`

`EndpointPropagateClient` starts and prompts for a message to send. Each message can be only one line long, and the client continues to prompt for a message until a `.` is entered as a message. The client then quits.

Each message entered into `EndpointPropagateClient` should appear in the output of `EndpointServer`. However, to truly see the effect of propagation, you might want to create a copy of the directory containing the `EndpointServer` code to run a second instance of `EndpointServer`. This enables you to see multiple peers receiving the message propagated by the client and illustrates the difference between propagation and the technique used by the example in the next section. When starting a second instance of `EndpointServer`, be sure to configure a different TCP and HTTP port for the JXTA platform.

Sending Messages Directly Using *EndpointMessenger*

The disadvantage of the propagation demonstrated in the previous example is that it's wasteful. Peers that might not be interested in the message receive the message, only to discard it. In the reference implementation, the TCP endpoint protocol implementation's use of TCP multicast limits this inefficiency to peers on the local LAN segment. To improve efficiency, it would be useful if a message could be sent to one specific peer using the `EndpointService` instance.

In fact, `EndpointService` does support this functionality through the `EndpointMessenger` interface. The `EndpointProtocol` interface allows a developer to obtain an `EndpointMessenger` instance for the endpoint protocol implementation. This object can be used to send messages to a specific peer located at a specific `EndpointAddress`. This functionality is used by the reference implementation to provide an implementation of the `OutputPipe` interface.

When starting `EndpointServer` in the "Using *EndpointServer* and *EndpointPropagateClient*" section, the `EndpointServer` prints the `EndpointAddress` for each protocol currently registered with the `EndpointService` instance. Each address follows the same basic format outlined in the "Endpoint Addresses" section earlier in this chapter. The example in Listing 9.3 prompts the user for a message and a destination address, and attempts to send the message using `EndpointMessenger`.

Listing 9.3 Source Code for *EndpointMessengerClient.java*

```
package com.newriders.jxta.chapter9;

import java.io.IOException;

import net.jxta.endpoint.EndpointAddress;
import net.jxta.endpoint.EndpointMessenger;
import net.jxta.endpoint.EndpointService;
import net.jxta.endpoint.Message;

import net.jxta.exception.PeerGroupException;
```

```
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;


/**
 * A simple Endpoint client that sends a message directly to a
 * service named EndpointServer concatenated with the Peer
 * Group ID, with service parameters 012345 on a specific peer
 * located at an Endpoint Address using an EndpointMessenger.
*/
public class EndpointMessengerClient
{
    /**
     * The peer group for the application.
     */
    private PeerGroup peerGroup = null;

    /**
     * The service name to use when listening for messages.
     * This service name will be appended with the Peer Group ID
     * of the peer group when the JXTA platform is started.
     */
    private String serviceName = "EndpointServer";

    /**
     * The service parameters to use when listening for
     * messages.
     */
    private String serviceParameters = "012345";


    /**
     * Starts the JXTA platform.
     *
     * @exception   PeerGroupException thrown if the platform
     *              can't be started.
```

```
 */
public void initializeJXTA() throws PeerGroupException
{
    peerGroup = PeerGroupFactory.newNetPeerGroup();

    // Add the Peer Group ID of the group to the service
    // name so the message is sent to the version of the
    // endpoint listener specific to this peer group.
    serviceName += peerGroup.getPeerGroupID().toString();
}

/**
 * Runs the application: starts the JXTA platform, accepts
 * user input message and endpoint info, and sends the
 * message to the Endpoint Address specified.
 *
 * @param   args the command-line arguments passed to the
 *     application.
 */
public static void main(String[] args)
{
    EndpointMessengerClient client =
        new EndpointMessengerClient();

    try
    {
        boolean done = false;
        String messageString = null;
        String addressString = null;

        // Initialize the JXTA platform.
        client.initializeJXTA();

        while (!done)
        {
            // Reset the strings.
```

```
            addressString = null;
            messageString = null;

            // Get the message; if the message is ., then
            // quit the application.
            System.out.print("Enter a message (or '.' "
                + " to quit): ");
            messageString = client.readInput();

            if ((messageString.length() > 0)
                && (!messageString.equals(".")))
            {
                // Get the destination Endpoint Address
                // from the user.
                System.out.print(
                    "Enter an endpoint address: ");
                while ((addressString == null)
                    || (addressString.length() == 0))
                {
                    addressString = client.readInput();
                }
                // Send a message to the server.
                client.sendMessage(
                    messageString, addressString);
            }
            else
            {
                // We're done.
                done = true;
            }
        }

        // Stop the JXTA platform. Currently, there isn't
        // any nice way to do this.
        System.exit(0);
    }
```

```
    catch (PeerGroupException e)
    {
        System.out.println("Error starting JXTA platform: "
            + e);
        System.exit(1);
    }
}

/**
 * Read a line of input from the system console.
 *
 * @return  the String read from the System.in InputStream.
 */
public String readInput()
{
    StringBuffer result = new StringBuffer();
    boolean done = false;
    int character;

    while (!done)
    {
        try
        {
            // Read a character.
            character = System.in.read();

            // Check to see if the character is a newline.
            if ((character == -1)
                || ((char) character == '\n'))
            {
                done = true;
            }
            else
            {
                // Add the character to the result string.
                result.append((char) character);
```

```
            }
        }
        catch (IOException e )
        {
            done = true;
        }
    }

    return result.toString().trim();
}

/**
 * Sends a message. In this case, the message string is sent
 * to the Endpoint Address specified, provided that the
 * Endpoint Address responds to a ping.
 *
 * @param   messageString the message to send to the peer.
 * @param   addressString the Endpoint Address of the
 *     destination peers.
 */
public void sendMessage(String messageString,
    String addressString)
{
    EndpointService endpoint =
        peerGroup.getEndpointService();
    EndpointAddress endpointAddress =
        endpoint.newEndpointAddress(addressString);

    // Manipulate the Endpoint Address to include the
    // appropriate destination service name and parameters.
    endpointAddress.setServiceName(serviceName);
    endpointAddress.setServiceParameter(serviceParameters);
    // Check that we can reach the Endpoint Address.
    if (endpoint.ping(endpointAddress))
    {
        // Create a new message.
```

```
            Message message = endpoint.newMessage();

            // Populate the message contents with the
            // messageString.
            message.setString("MessageText", messageString);

            try
            {
                EndpointMessenger messenger =
                    endpoint.getMessenger(endpointAddress);

                if (messenger != null)
                {
                    // Send the message directly to the Endpoint
                    // Address specified.
                    messenger.sendMessage(message);
                }
                else
                {
                    System.out.println("Unable to create "
                        + "messenger for given address.");
                }
            }
            catch (IOException e)
            {
                System.out.println("Error creating messenger "
                    + "or sending message: " + e);
            }
        }
        else
        {
            System.out.println("Unable to reach specified "
                + "address!");
        }
    }
}
```

Running the `EndpointMessengerClient` example requires similar steps to those outlined in the section "Using EndpointServer and EndpointPropagateClient":

1. Start an instance of `EndpointServer`.

2. Copy the `EndpointMessengerClient` source code into the same directory where you previously copied `EndpointPropagateClient`.

3. Compile `EndpointMessengerClient` using `javac -d . -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptixasn1.jar;instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;log4j.jar;minimalBC.jar EndpointMessengerClient.java`.

4. Start `EndpointMessengerClient` using `java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;instantp2p.jar;jxta.jar;jxtaptls. jar;jxtasecurity.jar;jxtashell.jar;log4j.jar;minimalBC.jar com.newriders.jxta.chapter9.EndpointMessengerClient`.

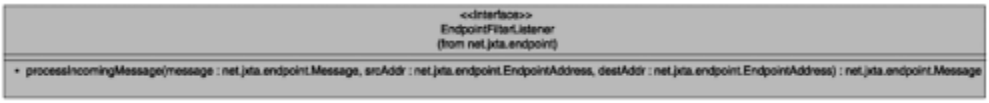
`EndpointMessengerClient` starts and prompts you to enter a message. After you have entered a message, the client prompts for a destination Endpoint Address. Enter one of the Endpoint Addresses printed by `EndpointServer` when it started.

The difference between `EndpointMessengerClient` and `EndpointPropagateClient` will become obvious if you copy the client directory and start a second instance of `EndpointServer`. Unlike in the `EndpointPropagateClient` example, only the server specified by the Endpoint Address entered into `EndpointMessengerClient` will receive the message.

The Endpoint Filter Listener

One other feature offered by the `EndpointService` interface is the capability to add filter listeners implementing the `EndpointFilterListener` interface (shown in Figure 9.6), defined in `net.jxta.endpoint`. `EndpointFilterListener` implementations can be registered with the `EndpointService` instance to allow a developer to arbitrarily preprocess incoming messages before they are handed off to the registered `EndpointListener` implementations.

Figure 9.6. The `EndpointFilterListener` interface.



Currently, the `EndpointFilterListener` interface is implemented by the `EndpointServiceStatsFilter` class in `net.jxta.impl.util`. This class is used to collect the message throughput statistics delivered by the Peer Information Protocol. The Rendezvous service reference implementation, `RendezVousServiceImpl`, also uses an inner class, `FilterListener`, to implement `EndpointFilterListener`. This implementation is used to prevent uncontrolled propagation and loopbacks.

Filter listeners are added to the Endpoint service using the `EndpointService.addFilterListener` method:

```
public void addFilterListener(String elementName,
    EndpointFilterListener listener, boolean incoming)
        throws IllegalArgumentException;
```

When registering a filter listener, the caller specifies whether the listener should be called to process incoming or outgoing messages. In addition, the caller specifies the name of a message element that a message must contain before the filter will be applied. Only those messages containing an element with a matching element name will have the filter applied to the message.

To understand how filters are applied to incoming messages, it is necessary to understand how incoming messages flow from an endpoint protocol implementation to registered `EndpointListener` instances. When an endpoint protocol implementation receives a complete message from a remote peer, it calls the `EndpointService.demux` method. The `demux` method implementation is responsible for first preprocessing the message using the registered `EndpointFilterListener` instances and then notifying registered `EndpointListener` instances. The `demux` method acts as a callback, freeing an endpoint protocol implementation from the duty of applying filters and notifying listeners itself.

In the current reference implementation, filters are not applied on outgoing messages. However, there is already some code in place, indicating that this feature will be implemented soon.

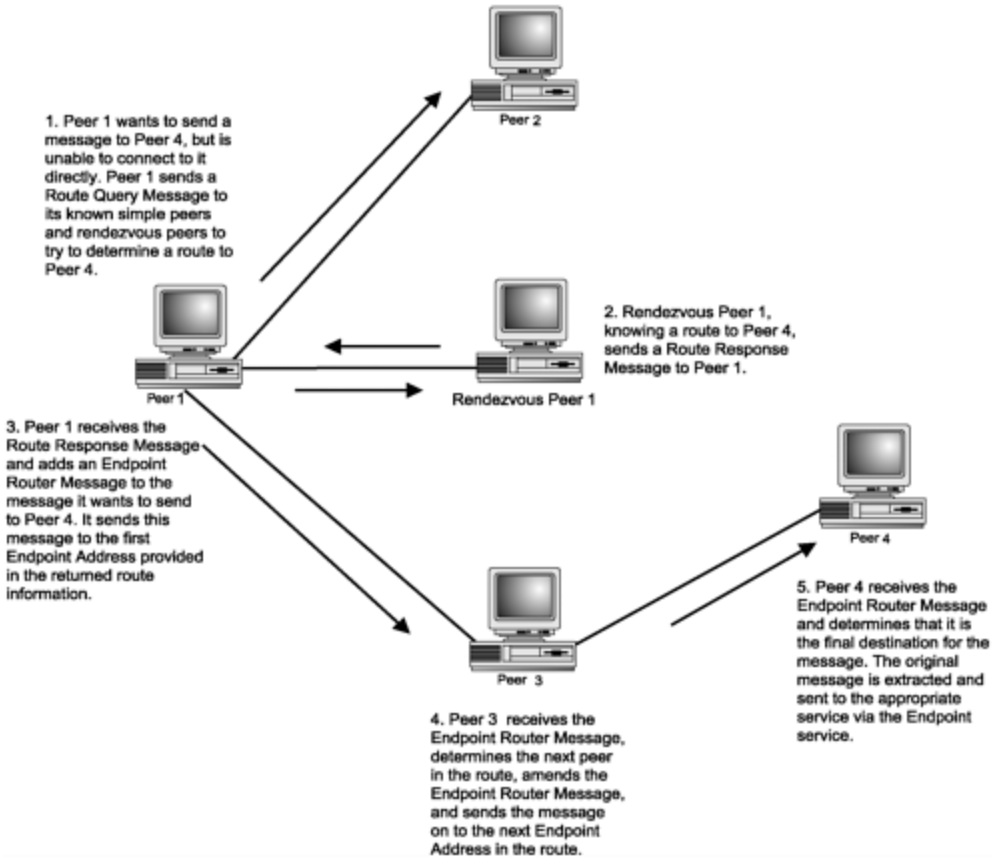
Although both `EndpointListener` and `EndpointFilterListener` define only a single `processIncomingMessage` method, there is one important difference between the two interfaces. Unlike `EndpointListener`, `EndpointFilterListener` 's version of `processIncomingMessage` returns a `Message` object. This object is used as input into subsequent filters and finally is used to either send the outgoing message to other peers or notify registered `EndpointListener` instances. If an `EndpointFilterListener` returns a `null` object, the message is discarded.

Introducing the Endpoint Routing Protocol

After examining the example code and the explanation of the `EndpointProtocol` and `EndpointMessenger` interfaces, you've probably realized that JXTA needs a mechanism to send messages between peers that aren't directly connected. Although the HTTP endpoint protocol implementation in the reference implementation provides router peer functionality that allows a message to traverse a firewall, a peer still needs some way to learn of the existence of the router peer in the first place. Because router peers may enter or leave the network spontaneously, a peer needs a routing mechanism that works even in situations in which the route between two peers is constantly changing. Enter the Endpoint Routing Protocol.

If two peers cannot communicate directly using a common endpoint protocol implementation, the Endpoint Routing Protocol provides each peer with a way to discover how it can send messages to the other peer via an intermediary, using only available endpoint protocol implementations (see Figure 9.7).

Figure 9.7. Flow of the Endpoint Routing Protocol.



The Endpoint Routing Protocol, also called the Peer Endpoint Protocol, provides a mechanism for a message to be sent to a remote peer using discovered route information. Each intermediary along the message route is responsible for passing the message on to the next peer described by the route information until the message reaches its ultimate destination.

For now, only two messages are required to determine route information: the Route Query Message and the Route Response Message. The current JXTA Protocols Specification defines three other messages for the Endpoint Routing Protocol: the Ping Query Message, the Ping Response Message, and the NACK Message. These messages allow a peer to test that a message can be routed to a destination peer and also allow an intermediary peer to signal the sender that an attempt to route a message has failed. These messages are not currently available in the reference implementation and will not be discussed.

The Endpoint Routing Protocol defines one other message, the Endpoint Router Message, which is used to pass route information along with a message. Peers along the message's path as it travels to its destination use the extra information provided by the Endpoint Router Message to determine the next peer en route to the destination.

The Route Query Message

A Route Query Message is sent by a peer when it wants to determine the set of ordered peers to use to send a message to a given Endpoint Address. Listing 9.4 shows the elements of the Router Query Message.

Listing 9.4 The Route Query Message XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:EndpointRouter>
  <Type>RouteQuery</Type>
  <DestPeer> . . . </DestPeer>
   <RoutingPeerAdv> . . . </RoutingPeerAdv>
</jxta:EndpointRouter>
```

Each element in the Route Query Message describes one aspect required to perform the search for route information:

- **`Type`**— A required element describing the type of Endpoint Router message being sent. For the Route Query Message, this element is set to `RouteQuery`.
- **`DestPeer`**— An optional element containing the Endpoint Address of the final destination peer in the route being discovered. Any route returned in response to this Route Query Message provides a route that allows a message to be sent from the local peer to the peer specified by `DestPeer`.
- **`RoutingPeerAdv`**— An optional element containing the Peer Advertisement of the peer requesting route information.

To discover route information, a peer sends a Route Query Message to other peers that it has previously discovered. In addition to finding peers by peer discovery, a peer may learn

of another peer's existence by processing a Route Query Message and extracting the `RoutingPeerAdv`, if one has been passed. By reusing this Peer Advertisement, the peer can save network bandwidth and potentially reduce the time required to obtain route information, resulting in improved performance.

As with any of the core protocols, a query might not result in a response or might result in multiple responses.

The Route Response Message

To provide a reply to a Route Query Message, a peer sends a Route Response Message describing a set of ordered Endpoint Addresses to use to send a message to a given destination peer. The Route Response Message has the format shown in Listing 9.5.

Listing 9.5 The Route Response Message XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:EndpointRouter>
  <Version>2</Version>
  <Type>RouteResponse</Type>
  <DestPeerIdTag> . . . </DestPeerIdTag>
  <RoutingPeerIdTag> . . . </RoutingPeerIdTag>
  <NbOfHops> . . . </NbOfHops>
  <RoutingPeerAdvTag> . . . </RoutingPeerAdvTag>
  <GatewayForward> . . . </GatewayForward>
</jxta:EndpointRouter>
```

The Route Response Message contains similar information to the Route Query Message, with the exception that it contains the route information requested by the peer:

- **`Version`**— A required element containing an integer describing the version of the Endpoint Routing Protocol being employed in the protocol conversation. Although the Protocols Specification defines this as a required element for both the Route Query and Response Message formats, the reference implementation currently adds it only in the Route Response Message. At this time, the `Version` is set to 2.

- **`Type`—** A required element containing a string describing the type of Endpoint Router Message being sent. For the Route Response Message, this element is set to `RouteResponse`.

- **`DestPeerIdTag`—** An optional element containing the Endpoint Address of the final destination peer in the route being discovered. This should match the Endpoint Address passed in the original Route Query Message's `DestPeerIdTag` element.

- **`RoutingPeerIdTag`—** An optional element containing the Endpoint Address of the peer that is acting as a source of route information. This will most likely be called `RoutingPeer` in the future because it contains an Endpoint Address rather than a Peer ID.

- **`RoutingPeerAdvTag`—** An optional element containing the Peer Advertisement of the peer requesting route information.

- **`NbOfHops`—** An optional element containing the number of network hops in the route to the destination peer.

- **`GatewayForward`—** An optional element containing the Endpoint Address of a peer along the route to the destination peer. There may be several `GatewayForward` elements in a Route Response Message, and the route depends on the order of these elements. The `GatewayForward` elements describe the path in order of Endpoint Addresses that a message must visit to reach a destination peer.

When a peer receives a Route Query Message, it checks to see if it knows how to route a message to the specified destination peer. If so, it returns the route information in a Route Response Message; otherwise, the current reference implementation discards the query and returns no response.

The Endpoint Router Message

The Endpoint Router Message provides the information required to route a message to its destination after the message has left its source peer. Rather than encapsulating the message being routed, the Endpoint Router Message simply adds routing information

alongside the other content of a message. The Endpoint Router Message provides the route information in the format in Listing 9.6.

Listing 9.6 The Endpoint Router Message XML

```
<jxta:JxtaEndpointRouter>
  <jxta:Src> . . . </jxta:Src>
  <jxta:Dest> . . . </jxta:Dest>
  <jxta:Last> . . . </jxta:Last>
  <jxta:NBOH> . . . </jxta:NBOH>
  <jxta:GatewayForward> . . . </jxta:GatewayForward>
  <jxta:GatewayReverse> . . . </jxta:GatewayReverse>
</jxta:JxtaEndpointRouter>
```

Each element provides information required to route a message to the destination peer and also how to route a response message to the source peer:

- **`Src`**— A required element containing the Endpoint Address of the original peer responsible for sending the message.
- **`Dest`**— A required element containing the Endpoint Address of the destination peer for the message.
- **`Last`**— An optional element containing the Endpoint Address of the previous peer in the routing order. This address corresponds to the peer responsible for sending a message to the current peer.
- **`NBOH`**— An optional element containing the number of network hops contained in the reverse route. If this parameter is set to `0`, it indicates that the message doesn't contain reverse routing information.
- **`GatewayForward`**— An optional element containing the Endpoint Address of a peer along the route to the destination peer. There may be several `GatewayForward` elements in a Route Response Message, and the route depends on the order of these elements. The `GatewayForward` elements describe the path in order of Endpoint Addresses that a message must visit to reach a destination peer.

- **`GatewayReverse`**— An optional element containing the Endpoint Address of a peer along the route from the destination peer to the source peer. There may be several `GatewayReverse` elements in a Route Response Message, and the reverse route depends on the order of these elements. The `GatewayReverse` elements describe the path in order of Endpoint Addresses that a message must visit to reach the original source peer.

As a peer receives an Endpoint Router Message, it determines the next peer in the route, modifies the Endpoint Router Message, and sends the message on to the next peer. The next peer in the route can be determined by either sending a Route Query Message or consulting the `GatewayForward` elements in the Endpoint Router Message accompanying the message.

Although a peer isn't required to populate the `GatewayForward` and `GatewayReverse` elements of the Endpoint Router Message before sending the message to the next peer, the JXTA Protocols Specification encourages peers to add this information. Adding this information not only reduces the processing and route query overhead required at each point along the route, but it also improves the performance of the routing process.

The Endpoint Router Transport Protocol

Up to this point, you might have assumed that the Endpoint Routing Protocol is implemented as a service, just like all the other core protocols in JXTA. However, to simplify the implementation of the Endpoint Routing Protocol, it is implemented as an endpoint protocol implementation, bound within the Endpoint service to the `jxta` protocol specifier. This endpoint protocol implementation, called the Endpoint Router Transport Protocol, is invoked when a message is sent to an Endpoint Address of the form `jxta://<Peer ID unique format>`. The mechanism is invoked in exactly the same fashion that the TCP endpoint protocol implementation gets invoked when sending a message to an Endpoint Address of the form `tcp://10.6.18.38`.

Because the Endpoint Router Transport Protocol is invoked automatically to handle messages being sent to Endpoint Addresses for the Endpoint Router, the developer never has to interact with the endpoint protocol implementation directly. To send a message to a remote peer via the Endpoint Router Transport, a developer needs only to create an Endpoint Router Endpoint Address from the Peer ID of the destination peer:

```
PeerID peerId;
EndpointServer endpoint;
. . .
String asString = "jxta://" + peerId.getUniqueValue().toString();
EndpointAddress address = endpoint.newEndpointAddress(asString);
```

After the `EndpointAddress` has been created, the service name and service parameters can be set, just as with any other Endpoint Address. The message can then be sent to the remote peer via the Endpoint Routing Transport Protocol by using the `EndpointService` to obtain an `EndpointMessenger` object for the `EndpointAddress`.

The Endpoint Router Transport Protocol in the reference implementation is provided by the `EndpointRouter` class in the `net.jxta.impl.endpoint` package. When the `getMessenger` method is called via the `EndpointService.getMessenger` method, `EndpointRouter` transparently handles determining route information, either from cached information or from sending Route Query Messages. If a direct connection is possible using one of the registered endpoint protocol implementations, the method returns the appropriate messenger; otherwise, the method returns an `EndpointRouter.EndpointRouterMessenger` object. This class implements `EndpointMessenger` and adds an Endpoint Router Message to outgoing message. The important point to realize here is that the Endpoint service is responsible for masking all of this from the developer. As long as the `jxta://` form of the Endpoint Address is used, the `EndpointService` instance handles the details of finding the right endpoint protocol implementation or routing information in a transparent fashion.

The Endpoint Routing Transport Protocol is incapable of propagating messages to multiple peers, and the reference implementation provides an empty implementation for `EndpointProtocol.propagate`. However, propagation isn't really the responsibility of the Endpoint Routing Transport Protocol. The Rendezvous service is responsible for propagating messages via known rendezvous peers when the peer is behind a firewall. Messages to individual rendezvous peers sent by the `RendezvousServiceImpl` use `EndpointMessenger`, allowing Endpoint Router–formatted Endpoint Addresses to correctly invoke the Endpoint Routing protocol implementation.

Summary

In this chapter, you explored the Endpoint service and the Endpoint Routing Protocol. These two elements are responsible for encapsulating and abstracting network transport-specific details and hiding those details from higher services. All that remains is to learn how to create new services and applications of your own. To do this, Chapter 10 discusses services and peer groups, how they relate, and how to create your own peer group and associate services with it.

Chapter 10. Peer Groups and Services

The concept of a peer group is central to all aspects of the JXTA platform. Peer groups provide a way of segmenting the P2P network space into distinct communities of peers organized for a specific purpose. All the core protocols explored in this book so far have depended on a peer group to provide the context for operations performed by services. Before creating a sample application, it is necessary to understand how JXTA peer groups are created and used to gain access to services.

This chapter provides the information you need to use peer groups and to understand how to create, join, or leave peer groups. Part of this coverage includes how JXTA allows peer groups to create private peer groups that are accessible to only authorized peers. In addition to the coverage of peer group semantics, the chapter explores how to create new services and create a peer group that can use these services.

Modules, Services, and Applications

Before diving into the specifics of working with peer groups, it is essential to understand the module framework employed by JXTA. The module framework is designed to allow a developer to provide functionality within JXTA in an extensible manner. Modules managed by the framework are responsible for providing all aspects of JXTA's functionality, including the implementation of the peer group mechanism as well as services and applications that are provided by a peer group.

Understanding modules is essential to being able to write new services for JXTA. To understand how these peer groups, services, and applications are specified, you first need to understand the JXTA concept of a module. Simply put, a *module* is some distributable chunk of functionality that a peer can initialize, start, and stop. In addition to a plain

module, JXTA provides the concept of a *service module*, a component used by a peer to run a service, and an *application module*, a component used by a peer to run an application. An application module is different from an application that invokes JXTA. An application that invokes JXTA can be comprised of several service and application modules.

To enable peers to discover modules, the definition of a module is divided into three types of advertisements: a Module Class Advertisement, a Module Specification Advertisement, and a Module Implementation Advertisement. Before diving into the specifics of each type of advertisement, it's important to understand how they are related.

Consider some of the problems inherent in creating a framework for modules in JXTA:

- Because JXTA is supposed to be language/platform agnostic, the module framework needs to support multiple implementations of a given module. For example, the Discovery service module might be implemented as a Java class or as a C++ COM class. Therefore, the framework needs to be capable of distinguishing among these module implementations, possibly using some external metadata representation, such as an advertisement.

- The capabilities of the module will invariably change over time. The person or organization responsible for defining the module might want to add or remove functionality, thus changing the specification of the module. For example, the Peer Discovery Protocol specification might change over time to add new search capabilities. Therefore, the module framework must be capable of distinguishing among various versions of a module, again using some metadata representation. In addition, each version of the module's specification might have multiple implementations, necessitating some link between the metadata describing a module implementation and the metadata describing the module's specification.

- There must be some way of referring to a module that provides a class of functionality independent of a particular specification or implementation of the module. For example, the JXTA Discovery service module is a class of module that provides discovery services. Again, there must be some relationship between the metadata describing a module specification and the metadata describing the module's class.

Each of these aspects is encapsulated by the Module Implementation, Module Specification, and Module Class Advertisements, respectively. The discussion of these advertisements in the following sections starts from the most general advertisement describing a module, the Module Class Advertisement.

The Module Class Advertisement

The first advertisement, the Module Class Advertisement, doesn't provide information on a module implementation; it exists solely to announce the existence of a class of module. The Module Class Advertisement provides only the few pieces of information shown in Listing 10.1.

Listing 10.1 The Module Class Advertisement XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:MCA>
  <MCID> . . . </MCID>
  <Name> . . . </Name>
  <Desc> . . . </Desc>
</jxta:MCA>
```

These pieces of information can be used by a peer to search for a module based on one of the advertisement's elements:

- **`MCID`**— A required element containing a Module Class ID. This ID uniquely identifies a class of modules. The Module Class ID is used as the basis for the IDs contained in the Module Specification and Implementation Advertisements.

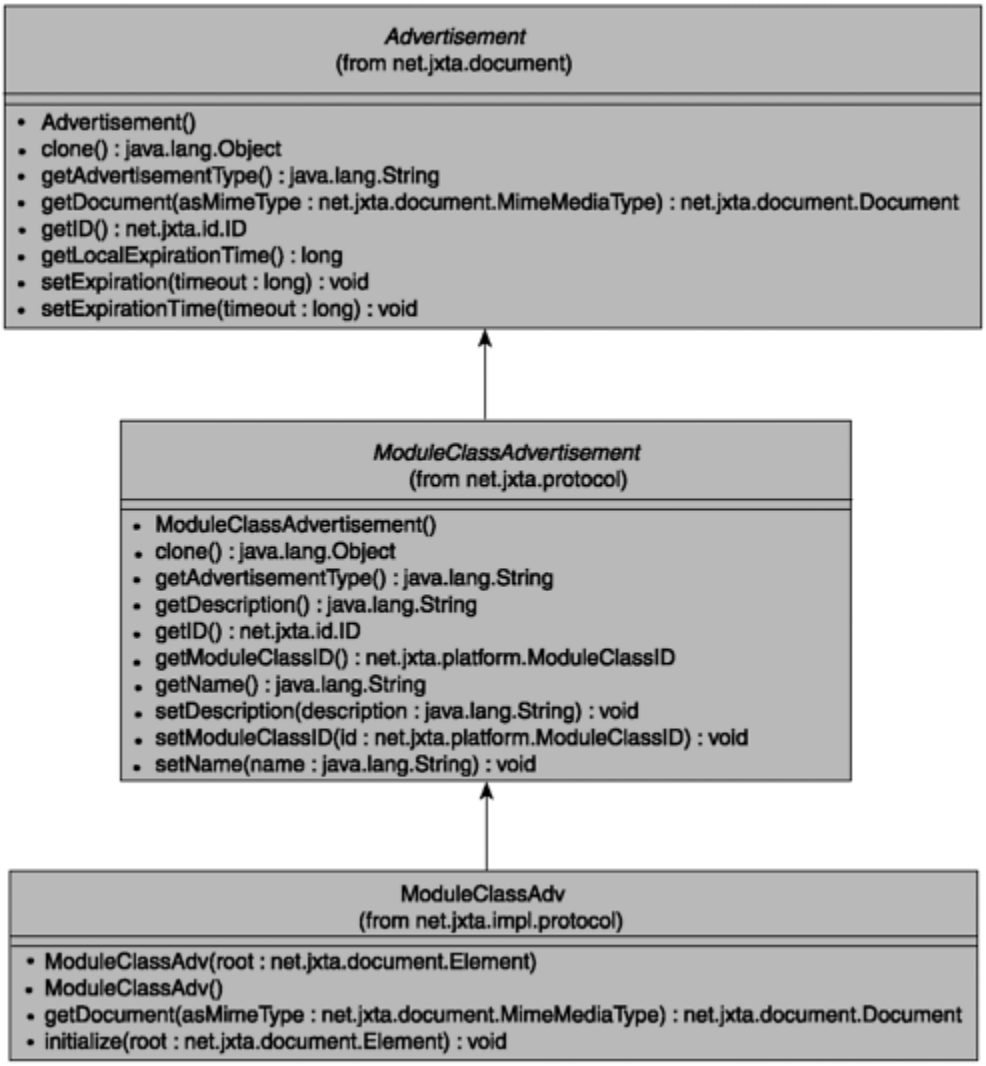
- **`Name`**— An optional element containing a simple name for the module class. This string is not necessarily unique.

- **`Desc`**— An optional element containing a description of the module class.

As with all other advertisement types in the reference implementation, the implementation of the Module Class Advertisement is split into the abstract definition `ModuleClassAdvertisement`, defined in `net.jxta.protocol`, and the reference

implementation `ModuleClassAdv`, defined in `net.jxta.impl.protocol`. These classes are shown in Figure 10.1.

Figure 10.1. The Module Class Advertisement classes.



Each class of modules has a unique Module Class Advertisement. For example, the reference implementation of the Discovery service is associated with a Module Class Advertisement that defines a class of modules responsible for providing discovery capabilities. Modules that provide this capability also use this same Module Class Advertisement. A different class of module, such as a module that provides routing capabilities, is associated with a different Module Class Advertisement.

The Module Specification Advertisement

The second advertisement responsible for defining a module is the Module Specification Advertisement. The purpose of a Module Specification Advertisement, shown in Listing 10.2, is to uniquely identify a set of protocolcompatible modules.

Listing 10.2 The Module Specification Advertisement XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:MSA>
  <MSID> . . . </MSID>
  <Name> . . . </Name>
  <Crtr> . . . </Crtr>
  <SURI> . . . </SURI>
  <Vers> . . . </Vers>
  <Desc> . . . </Desc>
  <Parm> . . . </Parm>
  <jxta:PipeAdvertisement> . . . </jxta:PipeAdvertisement>
  <Proxy> . . . </Proxy>
  <Auth> . . . </Auth>
</jxta:MSA>
```

The advertisement provides metadata on a version of the module's specification using the following elements:

- **`MSID`**— A required element containing a Module Specification ID that uniquely identifies the module specification. The Module Specification ID includes within it the Module Class ID identifying the class of module to which this specification belongs.

- **`Name`**— An optional element containing a simple name for the module specification. This string is not necessarily unique.

- **`Crtr`**— An optional element containing the name of the creator of the module specification.

- **`SURI`**— An optional element containing a URI that points to a specification document that describes the purpose and protocol, if any, defined by the module.

- **`Vers`**— A required element containing information on the specification version embodied by this Module Specification Advertisement.

- **`Desc`**— An optional element containing a description of the module specification.

- **`Parm`**— An optional element containing parameters for the specification. The format and meaning of these parameters is defined by the module's specification.

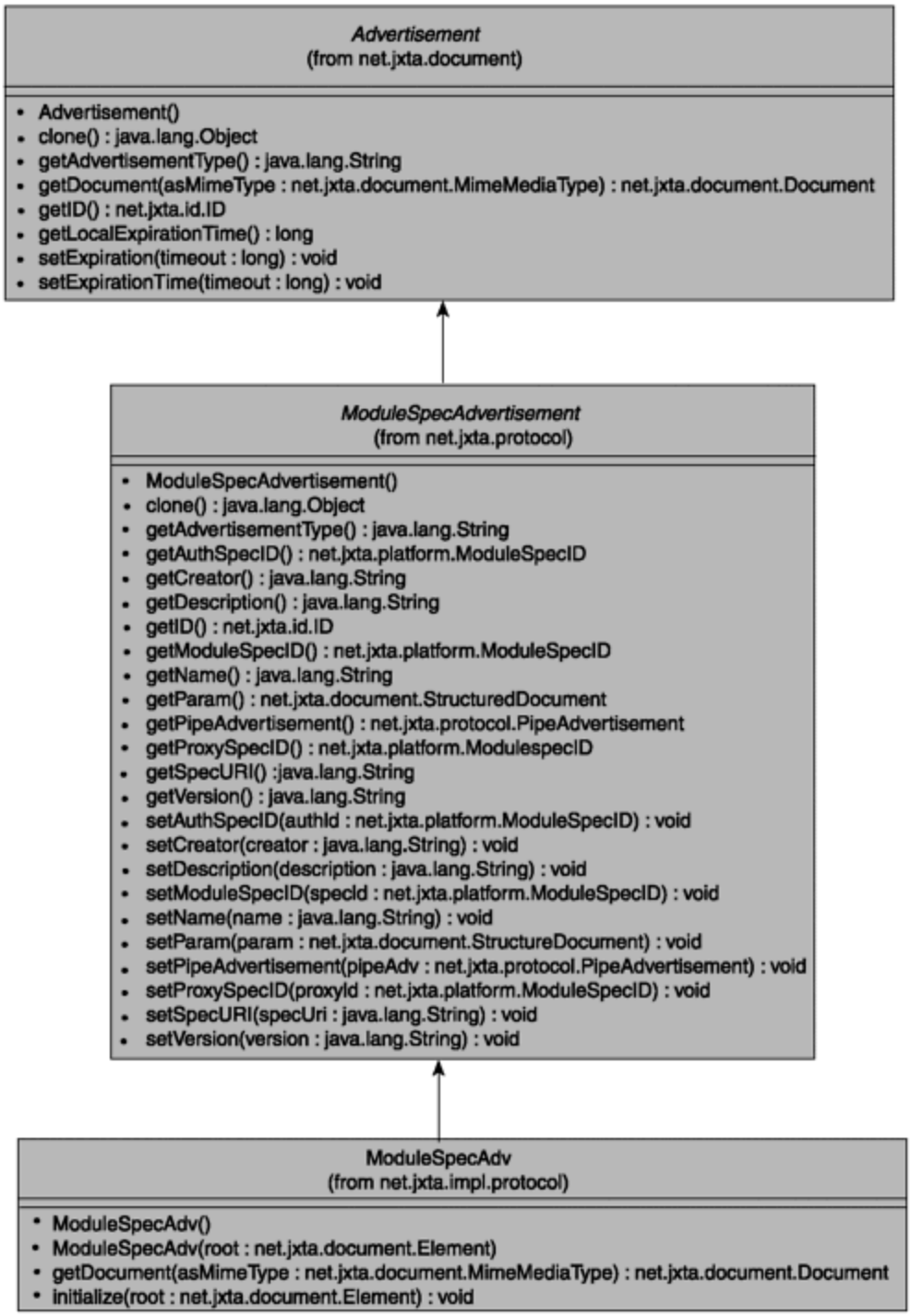
- **`jxta:PipeAdvertisement`**— An optional element containing a Pipe Advertisement describing a pipe that can be used to send data to the module. This element is actually the root element of the Pipe Advertisement, not an element that contains a Pipe Advertisement. The module that implements this specification binds an input pipe to the pipe identified by the Pipe Advertisement, allowing third parties to communicate with the module.

- **`Proxy`**— An optional element containing the Module Specification ID of a module that can be used to proxy communication with a module defined by this module specification. This is not really used in the reference implementation, and its use in modules is discouraged.

- **`Auth`**— An optional element containing the Module Specification ID of a module that provides authentication services for a module defined by this module specification.

The implementation of the Module Specification Advertisement is split into the abstract definition `ModuleSpecAdvertisement`, defined in `net.jxta.protocol`, and the reference implementation `ModuleSpecAdv`, defined in `net.jxta.impl.protocol`. These classes are shown in Figure 10.2.

Figure 10.2. The Module Specification Advertisement classes.



For every Module Class Advertisement, there can be one or more different Module Specification Advertisements, each specifying a different version of the module. For example, if a new version of the Peer Discovery Protocol is released, the module

responsible for implementing the PDP in the reference implementation will be associated with a new Module Specification Advertisement that identifies the new version of the protocol that it implements.

The Module Implementation Advertisement

The final advertisement responsible for defining a module, the Module Implementation Advertisement, provides information on a particular implementation of a module specification, as shown in Listing 10.3.

Listing 10.3 The Module Implementation Advertisement XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:MIA>
  <MSID> . . . </MSID>
  <Comp> . . . </Comp>
  <Code> . . . </Code>
  <PURI> . . . </PURI>
  <Prov> . . . </Prov>
  <Desc> . . . </Desc>
  <Parm> . . . </Parm>
</jxta:MIA>
```

The Module Implementation Advertisement provides information on a concrete implementation of a module specification:

- **`MSID`**— A required element containing the Module Specification ID identifying the module specification that this module is implementing.

- **`Comp`**— A required element containing compatibility information. The format of the information contained by this element depends on the possible deployment platforms for modules. Currently, the reference implementation defines an XML format for the compatibility information that details the JVM and binding. Future work to provide other bindings will result in a variety of formats for this information.

- **`Code`**— A required element containing any information required to run the code of the module implementation. The format of this information is again defined by the deployment platform in which the module will be running. Although this information is usually provided in addition to the package information provided by the `PURI`, the information in this element could provide the code for the implementation, eliminating the need for a `PURI` element.

- **`PURI`**— An optional element containing a URI that points to a package that contains the code responsible for providing the module implementation. In the reference implementation, the `Code` element provides the fully qualified class name for the module implementation, and the `PURI` element points to the location of a JAR file containing the class described by the `Code` element. Together, these elements can be used to download the module implementation if it doesn't exist locally and to start the module.

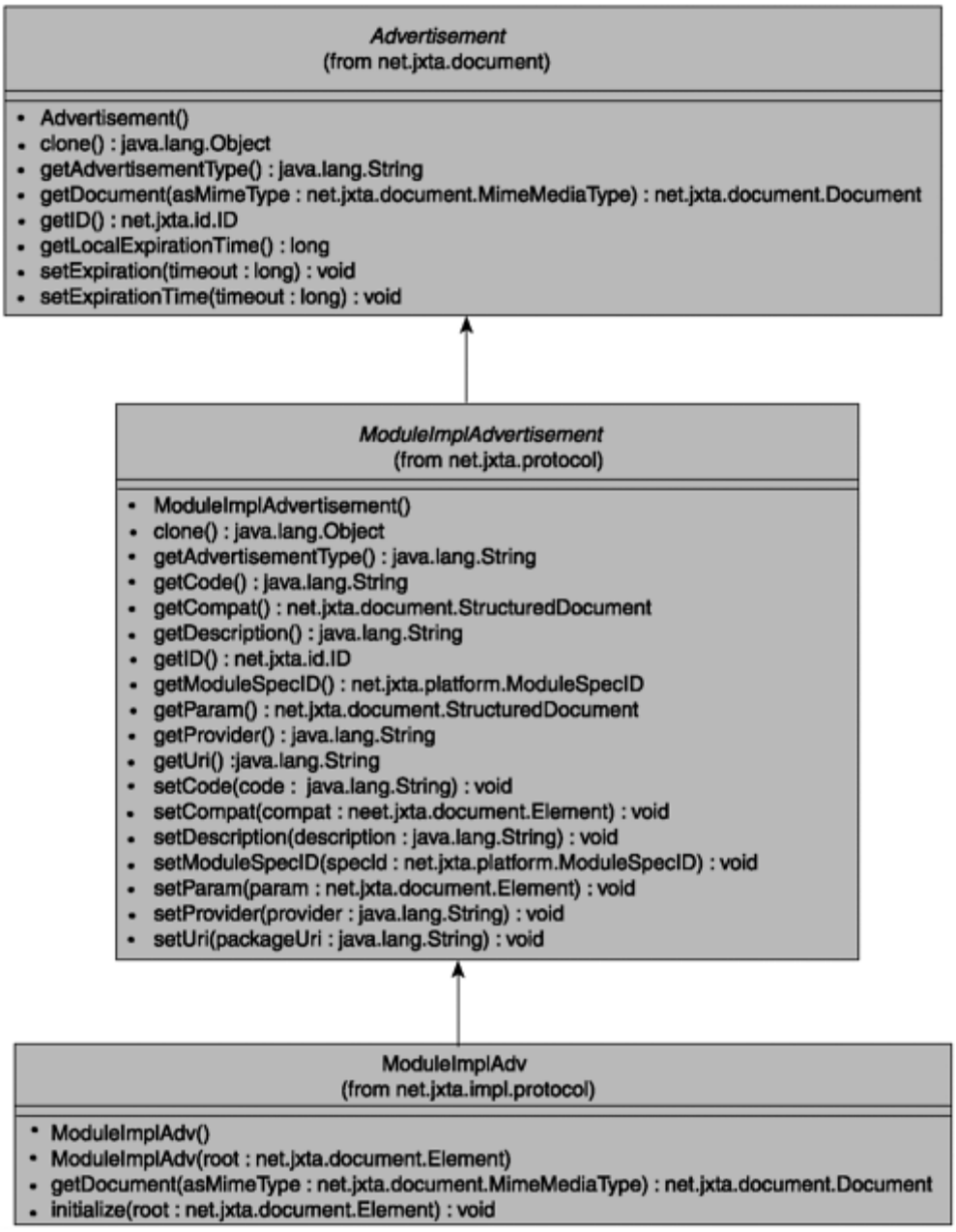
- **`Prov`**— An optional element containing the name of the entity that is providing the module implementation specified by this advertisement's `Code` or `PURI` elements.

- **`Desc`**— An optional element containing a description of the module implementation.

- **`Parm`**— An optional element containing parameters for the implementation. The format and meaning of these parameters is defined by the module's implementation.

The implementation of the Module Implementation Advertisement is similarly split into the abstract definition `ModuleImplAdvertisement`, defined in `net.jxta.protocol`, and the reference implementation `ModuleImplAdv`, defined in `net.jxta.impl.protocol`. These classes are shown in Figure 10.3.

Figure 10.3. The Module Implementation Advertisement classes.



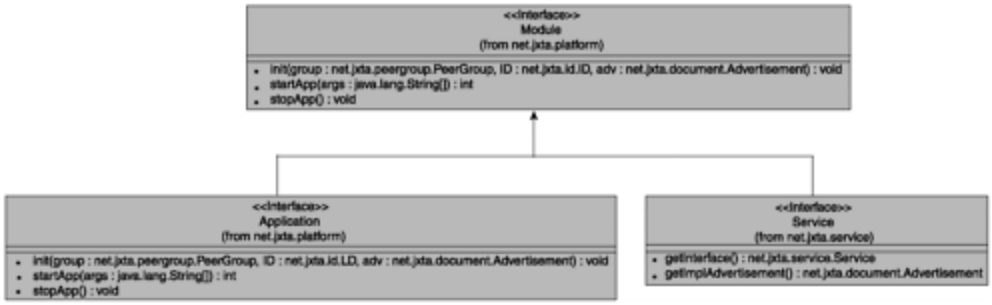
For every Module Specification Advertisement, there can be one or more different Module Implementation Advertisements, each specifying a different version of the module. Modules that are described by different module implementations that point to the same Module Specification Advertisement are compatible.

For example, if you have a C++ module and Java module each implementing the same version of the PDP, both will be associated with the same Module Specification Advertisement. Each implementation will be associated with different Module Implementation Advertisements that point to the same Module Specification Advertisement. The implementation-specific details, such as where to locate and download the module code, are specified in each module's Module Implementation Advertisement.

The Module, Service, and Application Interfaces

The actual implementation of a module can take one of three forms: a module, a service, or an application. For the most part, the functionality offered by each is almost identical, as are the interfaces that describe them, as shown in Figure 10.4.

Figure 10.4. The `Module`, `Service`, and `Application` interfaces.



The `Module` and `Application` interfaces are identical, providing the `init`, `startApp`, and `stopApp` methods. As their names suggest, these methods are used to initialize, start, and stop the module. Probably the most important of these three is the `init` method:

```
public void init(PeerGroup group, ID assignedID, Advertisement implAdv)
    throws PeerGroupException;
```

This method provides the module with a `PeerGroup` that it can use to obtain services, such as the Discovery or Resolver services. In addition, an `ID` for the module within the group is provided to allow the module to uniquely identify itself within the peer group. This `ID` can be used by modules as the root of their service name space, and it is usually used as the handler name that the module registers with the Resolver service. The `implAdv` parameter

is usually the `ModuleImplAdvertisement` that was used to instantiate the module, so it contains extra initialization parameters for the module in its `Parm` elements.

The `Service` interface adds only two additional methods: `getImplAdvertisement` and `getInterface`. The `getImplAdvertisement` method provides the Module Implementation Advertisement describing the service. The `getInterface` method returns another `Service` implementation that can be used to handle the `Service` implementation by proxy and protect the usage of the `Service`.

The Peer Group Lifecycle

To demonstrate the use of the JXTA reference implementation, the example code in this book has used one of two mechanisms to invoke the JXTA platform: command extensions running inside the JXTA Shell or a standalone application that invokes the platform directly. Each mechanism provided a way to access a `PeerGroup` object that allowed you to access services of a peer group.

In the case of the Shell, the examples started the platform using the `net.jxta.impl.peergroup.Boot` class:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar net.jxta.impl.peergroup.Boot
```

When the example Shell command extension ran, a `PeerGroup` object was obtained using the Shell's environment variables, as shown in Listing 10.4.

Listing 10.4 Obtaining the *PeerGroup* Object in a Shell Command

```
// Get the shell's environment.
theEnvironment = getEnv();

// Use the environment to obtain the current peer group.
ShellObject theShellObject = theEnvironment.get("stdgroup");
PeerGroup aPeerGroup = (PeerGroup) theShellObject.getObject();
```

When running the JXTA platform as a standalone application, the calling example applications have obtained a reference to a `PeerGroup` object using the following snippet of code:

```
net.jxta.peergroup.PeerGroup peerGroup =
    PeerGroupFactory.newNetPeerGroup();
```

Until now, no explanation has been given on why these mechanisms work or what goes on behind the scenes when either of these mechanisms is invoked. Each of these mechanisms is responsible for "bootstrapping" the JXTA platform, thereby preparing the platform to be used to perform P2P networking. This process revolves around instantiating two very special peer groups: the World Peer Group and the Net Peer Group. In JXTA, the peer group mechanism is implemented as a `Service` and therefore requires configuring the appropriate Module Implementation Advertisement. After this advertisement has been created, it is used to instantiate the Net Peer Group that allows the peer to communicate with other peers on the network.

Creating the World Peer Group

The first thing that the JXTA platform requires when bootstrapping is a *World Peer Group*, which is a peer group identified by a special Peer Group ID. The World Peer Group defines the basic capabilities of the peer, such as the services, endpoint protocol implementations, and applications that the peer will make available on the network.

Although each peer belongs to the World Peer Group, and the World Peer Group defines the endpoint protocol implementations supported by the peer, the World Peer Group itself can't be used to perform P2P networking. The World Peer Group is basically a template that can be used to either discover or generate a Net Peer Group instance. The Net Peer Group is a common peer group to peers in the network that allows all peers to communicate with each other.

In the reference implementation, the creation of the World Peer Group is managed by the `net.jxta.impl.peergroup.Platform` class. When bootstrapping the JXTA platform using either the `Boot` class or the `PeerGroupFactory.newNetPeerGroup` method, the `Platform` class is called to generate a Peer Advertisement and instantiate the World Peer

Group. Note that the `Platform` class is simply a special implementation of `PeerGroup` configured to handle the bootstrapping process. A different implementation can be provided by changing the `PlatformPeerGroupClassName` property in the `config.properties` file in the `net.jxta.impl` package.

The configuration information for the endpoint protocol implementations and other services supported by the World Peer Group is extracted from a Peer Advertisement used to configure the World Peer Group. In the reference implementation, the Peer Advertisement is generated by the `Configurator` tool based on configuration input provided by the user. The Peer Advertisement uses the format given in Listing 10.5.

Listing 10.5 The Peer Advertisement XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PA xmlns:jxta="http://jxta.org">
  <PID> . . . </PID>
  <GID> . . . </GID>
  <Name> . . . </Name>
  <Dbg> . . . </Dbg>
  <Desc> . . . </Desc>
  <Svc>
    <MCID> . . . </MCID>
    <Parm>
        . . .
    </Parm>
  </Svc>
</jxta:PA>
```

The parameters of the Peer Advertisement describe the fundamental information required for a remote peer to be capable of interacting with the peer:

- **`PID`—** A required element containing the Peer ID for the peer. The exact format of the ID used by JXTA isn't especially important for this discussion. The only important thing to note about the Peer ID at this point is that it incorporates the Peer Group ID in it. More information on the format of the ID used by the reference implementation can be found in the JXTA Protocols Specification.

- **GID**— An optional element containing the Peer Group ID of the peer group to which the peer described by this advertisement belongs.

- **Name**— An optional element containing a simple name for the peer. This string can be used in conjunction with the Discovery service to attempt to discover a particular peer; however, multiple peers may use the same `Name`. Only the Peer ID is guaranteed to uniquely identify a particular peer.

- **Dbg**— An optional element describing the debugging message level employed by the peer. This element is currently used only when configuring the peer while bootstrapping the platform. Currently accepted values for this element, from least explicit to most explicit, are `error`, `warn`, `info`, and `debug`.

- **Desc**— An optional element containing a description of the peer. As with the `Name` element, the contents of the `Desc` element are not necessarily unique among peers. This string can be used to perform discovery, with the same caveats as for the `Name` element.

- **Svc**— An optional element providing service configuration information. Note that multiple `Svc` elements may appear in the Peer Advertisement, each describing a different service. The format of the contents is unspecified by the Protocols Specification, and it is the responsibility of the `PeerGroup` implementation managing the bootstrapping process to know how to parse the `Svc` element's contents. The format expected by the reference implementation's `Configurator` class is a `MCID` element and a `Parm` element, explained next.

- **MCID**— The Module Class ID of the service that this `Svc` element is describing.

- **Parm**— The arbitrary parameters used to configure the service. The `Configurator` class understands only a few parameter formats, depending on the Module Class ID. The `Svc` parameters are mainly used to configure the peer's endpoint transports. Therefore, most parameters contain a Transport Advertisement containing configuration for the endpoint protocol implementation specified by the `MCID` element.

This configuration information is stored as a Peer Advertisement in a file called `PlatformConfig` in the current directory when the JXTA platform is started. Future

attempts to bootstrap the platform from the same directory will use the same `PlatformConfig` file to automatically configure the peer.

After finding or creating the Peer Advertisement, the `Platform` class is responsible for generating a Peer Group Advertisement that will be used to instantiate a World Peer Group. A Peer Group Advertisement is described using the format in Listing 10.6.

Listing 10.6 The Peer Group Advertisement XML

```
<?xml version="1.0" encoding="UTF-8"?>
<jxta:PGA xmlns:jxta="http://jxta.org">
  <GID> . . . </GID>
  <MSID> . . . </MSID>
  <Name> . . . </Name>
  <Desc> . . . </Desc>
  <Svc>
    . . .
  </Svc>
</jxta:PGA>
```

The elements of the Peer Group Advertisement provide the following information:

- **`GID`**— A required element containing a Peer Group ID that uniquely identifies this group. For the World Peer Group, this ID is the same on all peers.

- **`MSID`**— A required element containing a Module Specification ID that identifies the module providing the implementation of the peer group module on the peer. Modules will be discussed in the section "Creating the Net Peer Group," later in this chapter.

- **`Name`**— An optional element containing a simple name for the peer group. This string is not necessarily unique.

- **`Desc`**— An optional element containing a description of the peer group. The `Desc` and `Name` elements, like their counterparts in the Peer Advertisement, can be used to discover a Peer Group Advertisement.

- **`Svc`—** An optional element describing a service provided by the peer group. Note that multiple `Svc` elements may appear in the Peer Group Advertisement, each describing a different service. The format of the contents of the `Svc` is again dependent on the `PeerGroup` implementation.

As part of generating Peer Group Advertisement for the World Peer Group, the `Platform` object creates a Module Implementation Advertisement for the group. The Module Implementation Advertisement for the group describes the services offered by the peer group.

The Module Implementation Advertisement is populated with a set of hard-coded advertisements for the core platform services: the Discovery, Resolver, Rendezvous, Peer Info, and Endpoint services. In addition, elements containing advertisements for the endpoint protocol implementations are inserted into the advertisement. When the `Platform` instantiates the World Peer Group using the generated Peer Group Advertisement, the services and protocols described by the Module Implementation Advertisement are loaded and initialized by the `Platform`.

The sequence of events leading to the creation of a World Peer Group is triggered when the `Platform` is invoked by either the `Boot` class or `PeerGroupFactory.newNetPeerGroup()`. Each of these methods invokes the `Platform` class and uses the hard-coded set of services to instantiate a World Peer Group. In some applications, it might not be desirable to load all the services hard-coded into the `Platform` class. In this case, you can create a Peer Group Advertisement yourself, use it to create a `PeerGroup` instance, and pass the resulting `PeerGroup` as a parameter to the other version of `PeerGroupFactory.newNetPeerGroup`:

```
public static PeerGroup newNetPeerGroup(PeerGroup pg)
    throws PeerGroupException
```

This version of `newNetPeerGroup` bypasses the creation of a World Peer Group using the `Platform` class, using the provided `PeerGroup` instance as the World Peer Group instead, and proceeds directly to the creation of the Net Peer Group.

Creating the Net Peer Group

After the World Peer Group has been created, the peer needs to instantiate the Net Peer Group. The Net Peer Group can describe additional characteristics about a peer, but most often it is simply a duplicate of the World Peer Group. Although the Net Peer Group can be discovered, the majority of peers using the reference implementation rely on the version of the Net Peer Group's Peer Advertisement hard-coded into the World Peer Group advertisement produced by the `Platform`.

So what exactly is the Net Peer Group? Basically, the Net Peer Group is the peer's starting point on the P2P network. All peers belong to a Net Peer Group, but not necessarily the same Net Peer Group. For example, an enterprise application might define its own Net Peer Group that peers instantiate during the bootstrapping process. All members of the enterprise would be capable of using this Net Peer Group to communicate, but other JXTA peers wouldn't be capable of communicating with this network by default.

The World Peer Group is different from the Net Peer Group in that it is really only a configuration mechanism. The Net Peer Group is the peer group used to provide actual connectivity to the P2P network.

To generate the Net Peer Group, the `Platform` class hard-codes the `StartNetPeerGroup` application in `net.jxta.impl.peergroup` into the World Peer Group's Module Implementation Advertisement. This application is invoked by the platform and causes the `StartNetPeerGroup` class to discover or build the Net Peer Group's Peer Advertisement. Although the `StartNetPeerGroup` does provide a way for a user to choose to discover the Net Peer Group, this functionality is fairly hidden from the user. Usually, the `StartNetPeerGroup` builds a Peer Group Advertisement from the parent World Peer Group using a default Peer Group ID for the Net Peer Group.

After the Net Peer Group is instantiated, the services described by the peer group's Module Implementation Advertisement are started. In the standard Net Peer Group, this forces the core services to begin providing the Discovery, Resolver, Rendezvous, Peer Info, and Endpoint services. After these services are started, the peer is connected to the network and ready to interact with other peers. New peer groups can be created to segment the network space, using the Net Peer Group as a template for the set of services offered by the peer group.

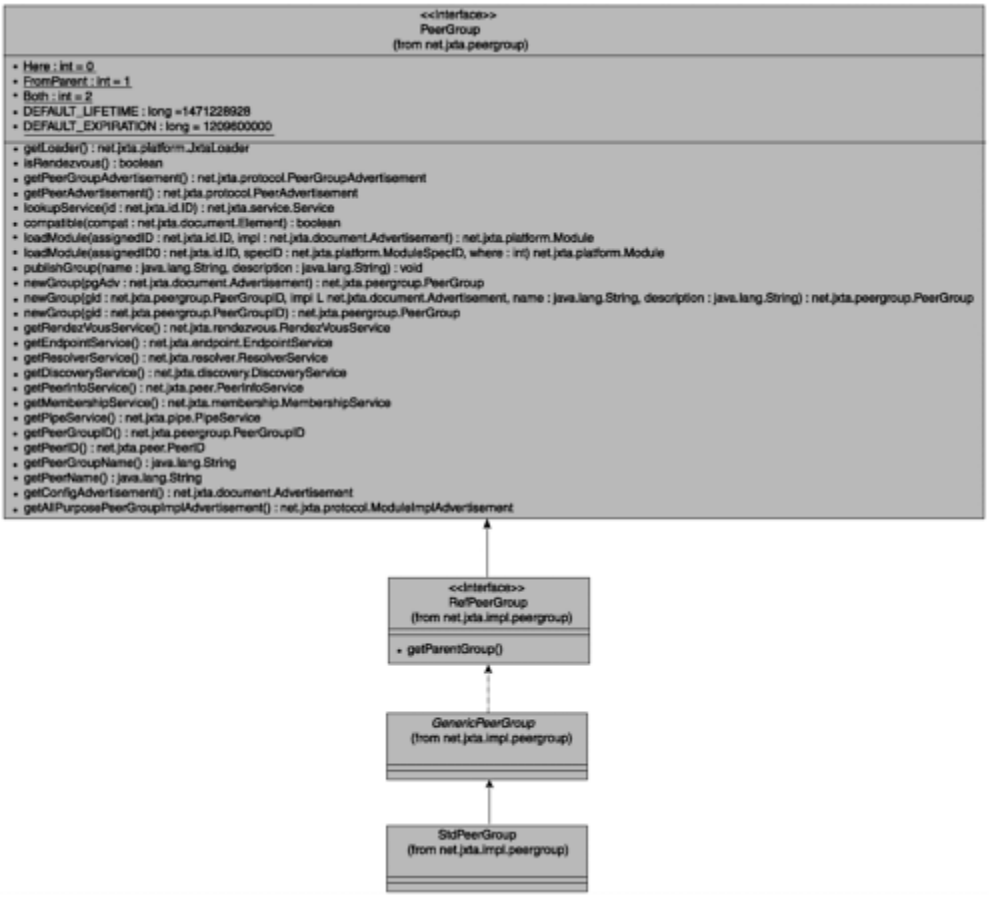
In summary, the process of bootstrapping instantiates the World Peer Group, which comprises a set of services that are usually hard-coded into the JXTA platform implementation. After the World Peer Group has been instantiated, the peer uses the services of the World Peer Group to discover or generate a Net Peer Group instance. The Net Peer Group instance provides access to the P2P network and a set of core services used by the peer. After booting into the Net Peer Group, the peer can use the services offered by the Net Peer Group. The peer may also elect to create other peer groups with extra capabilities by using the Net Peer Group as a template for the set of core services offered by the peer group.

Working with Peer Groups

After the World and Net Peer Groups have been created, peers can communicate with each other using the core JXTA protocols. However, the Net Peer Group provides a common space where everyone in the P2P network can interact, a property that might not be suitable to all P2P applications. To allow peers to group themselves in some meaningful way, peers can form their own peer groups, each providing its own set of services to members of the peer group.

Working with peer groups requires use of the `PeerGroup` interface, shown in Figure 10.5, defined in `net.jxta.peergroup`, and its implementations. The `StdPeerGroup` class defined in `net.jxta.impl.peergroup` provides the `PeerGroup` implementation used throughout the reference implementation.

Figure 10.5. The `PeerGroup` interface and implementation classes.



The `PeerGroup` interface defines the standard Module Class IDs for the core JXTA services and Module Specification IDs for the reference implementations of those core services.

Creating a Peer Group

Creating a peer group isn't much different from using any of the other services that a peer group provides. To create a peer group, you only need to call one of the `newGroup` methods, shown in Listing 10.7, provided by the `PeerGroup` interface. Each of the different versions has a slightly different set of circumstances under which it should be invoked.

Listing 10.7 Peer Group Creation Methods

```
public PeerGroup newGroup(Advertisement pgAdv)
```

```
    throws PeerGroupException;newGroup(Advertisement);
public PeerGroup newGroup(PeerGroupID gid)
    throws PeerGroupException;
public PeerGroup newGroup(PeerGroupID gid, Advertisement impl,
    String name, String description)
        throws PeerGroupException;
```

The first version of `newGroup` uses a given Peer Group Advertisement to instantiate the peer group. This version is used to create a peer group using an existing Module Implementation Advertisement.

The second version of `newGroup` creates a new peer group using the given Peer Group ID. The version assumes that a Peer Group Advertisement with the corresponding Peer Group ID has already been published.

The final version of `newGroup` creates a new peer group using the given Peer Group ID, Module Implementation Advertisement, name, and description. If the given `PeerGroupID` is `null`, this method creates a new Peer Group ID for the new group.

The example in Listing 10.8 shows how to create a new peer group using the Net Peer Group and the `newGroup` method. This version simply makes a copy of the Net Peer Group's Module Implementation Advertisement and uses it to instantiate the new group. The `newGroup` method also publishes the Peer Group Advertisement for the new peer group in the parent peer group. The parent peer group is considered to be the peer group used to create the group through the call to `newGroup`.

Listing 10.8 Source Code for *CreatePeerGroup.java*

```
package com.newriders.jxta.chapter10;

import java.util.Enumeration;

import net.jxta.discovery.DiscoveryService;

import net.jxta.exception.PeerGroupException;
import net.jxta.id.IDFactory;
```

```
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.peergroup.PeerGroupFactory;

import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;


/**
 * Create a new peer group and publish it.
 */
public class CreatePeerGroup
{
    /**
     * The Net Peer Group for the application.
     */
    private PeerGroup netPeerGroup = null;


    /**
     * Creates a new peer group using the Net Peer Group's
     * Module Implementation Advertisement as a template.
     *
     * @exception   Exception if an error occurs retrieving the
     *               copy of the Net Peer Group's Module
     *               Implementation Advertisement.
     */
    public void createPeerGroup() throws Exception
    {
        // The name and description for the peer group.
        String name = "CreatePeerGroup";
        String description =
            "An example peer group to test peer group creation";

        // Obtain a preformed ModuleImplAdvertisement to
        // use when creating the new peer group.
```

```
    ModuleImplAdvertisement implAdv =
        netPeerGroup.getAllPurposePeerGroupImplAdvertisement();

    // Create the Peer Group ID.
    PeerGroupID groupID = IDFactory.newPeerGroupID();
    // Create the new group using the Peer Group ID,
    // advertisement, name, and description.
    PeerGroup newGroup = netPeerGroup.newGroup(
        groupID, implAdv, name, description);

    // Need to publish the group remotely only because
    // newGroup() handles publishing to the local peer.
    PeerGroupAdvertisement groupAdv =
        newGroup.getPeerGroupAdvertisement();
    DiscoveryService discovery =
        netPeerGroup.getDiscoveryService();
    discovery.remotePublish(groupAdv,
        DiscoveryService.GROUP);
}

/**
 * Starts the JXTA platform.
 *
 * @exception   PeerGroupException thrown if the platform
 *              can't be started.
 */
public void initializeJXTA() throws PeerGroupException
{
    netPeerGroup = PeerGroupFactory.newNetPeerGroup();
}

/**
 * Runs the application.
 *
 * @param   args the command-line arguments passed to
 *     the application.
```

```
     */
    public static void main(String[] args)
    {
        CreatePeerGroup creator = new CreatePeerGroup();

        try
        {
            // Initialize the JXTA platform.
            creator.initializeJXTA();

            // Create the group.
            creator.createPeerGroup();
            // Exit.
            System.exit(0);
        }
        catch (Exception e)
        {
            System.out.println("Error starting JXTA platform: "
                + e);
            System.exit(1);
        }
    }
}
```

The `CreatePeerGroup` example doesn't really do much of note, but it provides the first step toward creating a new peer group that provides a new service.

Joining a Peer Group

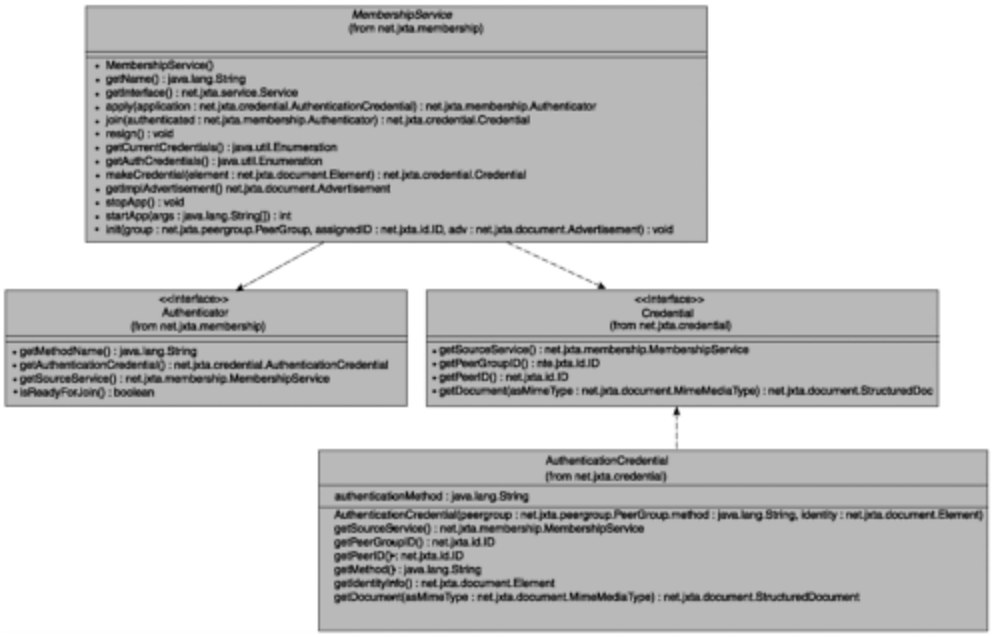
Instantiating a peer group from an advertisement is only the first step toward being able to interact with members of the peer group. Before a peer can interact with the group, it needs to join the peer group, a process that allows the peer to establish its identity within the peer group. This process allows peer groups to permit only authorized peers to join and interact with the peer group.

Each peer group has a membership policy that governs who can join the peer group. When a peer instantiates a peer group, the peer group's membership policy establishes a temporary identity for the peer, similar to the "nobody" identity in UNIX systems. This temporary identity exists for the sole purpose of allowing the peer to establish its identity by interacting with the membership policy. This interaction can involve the exchange of login information, exchange of public keys, or any other mechanism that a peer group's membership implementation uses to establish a peer's identity.

After a peer has successfully established its identity within the peer group, the membership policy provides the user with credentials. These credentials can then be used to provide verification of identity to other peers in the group when interacting with services offered by the peer group.

The membership policy for a peer group is implemented as the Membership service. The Membership service in the reference implementation is defined by `MembershipService`, shown in Figure 10.6, which is part of the `net.jxta.membership` package.

Figure 10.6. The Membership service and related classes.



In addition to the `MembershipService` class and its implementations, the reference implementation defines a `Credential` interface and an implementation called

`AuthenticationCredential`. These classes, defined in `net.jxta.credential`, are used in conjunction with the `MembershipService` class to represent an identity and the access level associated with that identity.

Two steps are involved in establishing an identity within a peer group using the peer group's `MembershipService` instance:

1. **Applying for membership.** This involves calling the peer group's `MembershipService` 's `apply` method. The `apply` method takes an `AuthenticationCredential` argument, which specifies the authentication method and desired identity. The method returns an `Authenticator` implementation that the caller can use to authenticate with the peer group.

2. **Joining the group.** The peer must provide the `Authenticator` implementation with the information that it requires to authenticate. When the peer has completed authentication using the `Authenticator`, the `Authenticator` 's `isReadyForJoin` method returns `true`. The peer now calls the `MembershipService` 's `join` method, providing the `Authenticator` as an argument. The `join` method returns a `Credential` object that the peer can use to prove its identity to peer group services.

When applying for membership, the peer making the request must know the implementation of `Authenticator` to interact with the `Authenticator`. This is required because the `Authenticator` interface has no mechanism for the peer to interact with it. Using only the `Authenticator` interface, a peer can only determine whether it has completed the authentication process successfully and then proceed with joining the peer group.

The reference implementation currently provides two example `MembershipService` implementations in the `net.jxta.impl.membership` package: `NullMembershipService` and `PasswdMembershipService`. `NullMembershipService` provides a `MembershipService` that offers no real authentication and simply assigns whatever identity the peer requests. `PasswdMembershipService` provides a simple authentication based on login and "encrypted" passwords provided in the parameters in the advertisement for the Membership service. The "encryption" consists of a simple substitution cipher and thus is not practical for securing a real peer group.

Leaving a Peer Group

To leave a peer group, the peer simply calls the `resign` method on the `MembershipService` implementation for the peer group. This removes all authentication credentials from the `MembershipService` and reverts the peer to the "nobody" identity within the peer group.

The Current Membership Implementation

Unfortunately, the current implementations of `MembershipService` aren't especially useful. To provide proper authentication, a developer must develop a `MembershipService` of his own to manage the creation and validation of authentication credentials. In addition, the developer must provide a mechanism in his service to use the credentials and validate the credentials passed by other peers in requests to the service.

Although the Protocol Specification outlines the concept of an Access service whose responsibility it is to verify credentials passed with requests, no implementation is provided in the reference implementation. The Resolver service's Resolver Query and Response Messages support a `Credential` element, but the contents of the element are never verified. For now, it appears that it is the responsibility of a developer to define his own Access service implementation and use it to verify credentials passed to his custom peer group services.

As such, a developer currently needs only to instantiate a peer group and can skip the steps of applying for membership and joining the peer group. This will undoubtedly change in the future, but for now, you can safely ignore the Membership service.

Destroying a Peer Group

Many people ask, "How do I destroy a peer group that I created?" Unfortunately, there is no way to explicitly destroy a peer group after it has been created and its advertisement has been published. Although the `PeerGroup` instance on a particular peer might be destroyed, other peers on the network can still instantiate a `PeerGroup` object for the group as long as they can find the Peer Group Advertisement for the group. After the Peer Group

Advertisement has been published, the peer group exists in the network until the advertisement expires.

If the Peer Group Advertisement was published to other peers using the default lifetime, the advertisement is removed from other peers' caches after two hours. However, if the Peer Group Advertisement was published locally using the default lifetime, it will not expire for a year.

One solution to this problem is to publish Peer Group Advertisements using a short lifespan. That way, the peer group will expire quickly if the advertisement isn't being used. However, doing this requires not using the `PeerGroup.publishGroup` method or the `PeerGroup.newGroup` method. By default, the `PeerGroup` reference implementation's `newGroup` method will call `publishGroup` to publish the Peer Group Advertisement. To publish the Peer Group Advertisement with a nondefault expiration or lifespan, you must manually create the `PeerGroup` instance and publish the Peer Group Advertisement using the following steps:

1. Create the `PeerGroupAdvertisement` instance using the `AdvertisementFactory.newAdvertisement` method, passing in the `String` obtained by calling the static `PeerGroupAdvertisement.getAdvertisementType` method.

2. Populate the fields of the `PeerGroupAdvertisement` instance, making sure to generate a new Peer Group ID using a call to `IDFactory` 's `newPeerGroupID` method.

3. Load the `PeerGroup` instance from the advertisement by calling the parent `PeerGroup` 's `loadModule` method, passing in the Peer Group ID from the Peer Group Advertisement and the Module Implementation Advertisement for the new Peer Group.

4. Publish the Peer Group Advertisement locally and remotely using the parent `PeerGroup` instance's `DiscoveryService` instance. This enables you to set the expiration and lifespan for the Peer Group Advertisement.

Creating a Service

Creating a service is a fairly simple task: Create a class that implements the `Service` interface. In addition to creating the `Service` implementation itself, other parts make up a good service design. A good service design separates, abstracts, and encapsulates the elements of the implementation in an object-oriented fashion.

The example service presented in the following sections extends one of the Resolver service examples presented in Chapter 5, "The Peer Resolver Protocol." The example given in that chapter showed how to use the Resolver service to create a `QueryHandler` that can handle a custom request that poses a basic math problem: What is the value of the given base raised to the given power? This example extends the basic functionality provided by that `QueryHandler` example and builds a full-fledged service module.

The Example Service Messages

To simplify the implementation of the example service, the example in Listing 10.9 reuses the `ExampleQueryMsg` and `ExampleResponseMsg` classes created in Chapter 5. These classes provide the functionality required to parse and format the XML used by the `QueryHandler` to send a query and receive a response.

Listing 10.9 Source Code for *ExampleQueryMsg.java*

```
package com.newriders.jxta.chapter10;

import java.io.InputStream;
import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.Document;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
```

```
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;


/**
 * An example query message, which will be wrapped by a
 * Resolver query message to send the query to other peers.
 * The query essentially asks a simple math question: "What
 * is the value of (base) raised to (power)?"
 */
public class ExampleQueryMsg
{
    /**
     * The base for query.
     */
    private double base = 0.0;

    /**
     * The power for the query.
     */
    private double power = 0.0;


    /**
     * Creates a query object using the given base and power.
     *
     * @param   aBase the base for the query.
     * @param   aPower the power for the query.
     */
    public ExampleQueryMsg(double aBase, double aPower)
    {
        super();

        this.base = aBase;
        this.power = aPower;
    }
```

```
    /**
     * Creates a query object by parsing the given input stream.
     *
     * @param      stream the InputStream source of the
     *             query data.
     * @exception  Exception if the message can't be parsed
     *             from the stream.
     */
    public ExampleQueryMsg(InputStream stream) throws Exception
    {
        StructuredTextDocument document = (StructuredTextDocument)
            StructuredDocumentFactory.newStructuredDocument(
                new MimeMediaType("text/xml"), stream);

        Enumeration elements = document.getChildren();

        while (elements.hasMoreElements())
        {
            TextElement element = (TextElement) elements.nextElement();

            if(element.getName().equals("base"))
            {
                base =
Double.valueOf(element.getTextValue()).doubleValue();
                continue;
            }

            if(element.getName().equals("power"))
            {
                power = Double.valueOf(
                    element.getTextValue()).doubleValue();
                continue;
            }
        }
    }
```

```
/**
 * Returns the base for the query.
 *
 * @return  the base value for the query.
 */
public double getBase()
{
    return base;
}

/**
 * Returns a Document representation of the query.
 *
 * @param      asMimeType the desired MIME type
 *             representation for the query.
 * @return     a Document form of the query in the
 *             specified MIME representation.
 * @exception    Exception if the document can't be created.
 */
public Document getDocument(MimeMediaType asMimeType)
    throws Exception
{
    StructuredDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            asMimeType, "example:ExampleQuery");
    Element element;

    element = document.createElement("base",
        Double.toString(getBase()));
    document.appendChild(element);

    element = document.createElement("power",
        Double.toString(getPower()));
    document.appendChild(element);
```

```
        return document;
    }

    /**
     * Returns the power for the query.
     *
     * @return  the power value for the query.
     */
    public double getPower()
    {
        return power;
    }

    /**
     * Returns an XML String representation of the query.
     *
     * @return  the XML String representing this query.
     */
    public String toString()
    {
        try
        {
            StringWriter out = new StringWriter();
            StructuredTextDocument doc =
                (StructuredTextDocument) getDocument(
                    new MimeMediaType("text/xml"));
            doc.sendToWriter(out);

            return out.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

`ExampleQueryMsg` is a simple class that wraps a base and power value for the exponentiation as an XML query that can be sent to another peer. The response to an `ExampleQueryMsg`, `ExampleResponseMsg`, contains the base and power values sent by the query, plus the result of the exponentiation. The source code for `ExampleResponseMsg` is shown in Listing 10.10.

Listing 10.10 Source Code for *ExampleResponseMsg.java*

```
package com.newriders.jxta.chapter10;

import java.io.InputStream;
import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.Document;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;



/**
 * An example query response, which will be wrapped by a Resolver response
 * message to send the response to the query. The response contains the
 * answer to the simple math question posed by the query.
 */
public class ExampleResponseMsg
{
    /**
     * The base from the original query.
     */
    private double base = 0.0;
```

```
    /**
     * The power from the original query.
     */
    private double power = 0.0;

    /**
     * The answer value for the response.
     */
    private double answer = 0;


     /**
     * Creates a response object using the given answer value.
     *
     * @param   anAnswer the answer for the response.
     */
    public ExampleResponseMsg(double aBase, double aPower, double
anAnswer)
    {
        this.base = aBase;
        this.power = aPower;
        this.answer = anAnswer;
    }

    /**
     * Creates a response object by parsing the given input stream.
     *
     * @param        stream the InputStream source of the response data.
     * @exception    Exception if the message can't be parsed from the
     *               stream.
     */
    public ExampleResponseMsg(InputStream stream) throws Exception
    {
        StructuredTextDocument document = (StructuredTextDocument)
            StructuredDocumentFactory.newStructuredDocument(
                new MimeMediaType("text/xml"), stream);
```

```
        Enumeration elements = document.getChildren();

        while (elements.hasMoreElements())
        {
            TextElement element = (TextElement) elements.nextElement();

            if(element.getName().equals("answer"))
            {
                answer = Double.valueOf(
                    element.getTextValue()).doubleValue();
                continue;
            }

            if(element.getName().equals("base"))
            {
                base =
Double.valueOf(element.getTextValue()).doubleValue();
                continue;
            }

            if(element.getName().equals("power"))
            {
                power = Double.valueOf(
                    element.getTextValue()).doubleValue();
                continue;
            }
        }
    }

    /**
     * Returns the answer for the response.
     *
     * @return  the answer value for the response.
     */
    public double getAnswer()
```

```
{
    return answer;
}
/**
 * Returns the base for the query.
 *
 * @return  the base value for the query.
 */
public double getBase()
{
    return base;
}

/**
 * Returns a Document representation of the response.
 *
 * @param       asMimeType the desired MIME type representation for
 *              the response.
 * @return      a Document form of the response in the specified
 *              MIME representation.
 * @exception   Exception if the document can't be created.
 */
public Document getDocument(MimeMediaType asMimeType) throws Exception
{
    Element element;
    StructuredDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            asMimeType, "example:ExampleResponse");

    element = document.createElement("base",
        Double.toString(getBase()));
    document.appendChild(element);

    element = document.createElement("power",
        Double.toString(getPower()));
    document.appendChild(element);
```

```
    element = document.createElement("answer", (
        new Double(getAnswer()).toString()));
    document.appendChild(element);

    return document;
}
/**
 * Returns the power for the query.
 *
 * @return  the power value for the query.
 */
public double getPower()
{
    return power;
}

/**
 * Returns an XML String representation of the response.
 *
 * @return  the XML String representing this response.
 */
public String toString()
{
    try
    {
        StringWriter out = new StringWriter();
        StructuredTextDocument doc = (StructuredTextDocument)
            getDocument(new MimeMediaType("text/xml"));
        doc.sendToWriter(out);

        return out.toString();
    }
    catch (Exception e)
    {
        return "";
```

```
        }
    }
}
```

The XML produced by these two classes, as well as the reasoning for encapsulating the classes, can be found in the original example in Chapter 5.

Creating a Listener Interface

The services throughout this book have employed listener objects to provide an elegant way for third-party developers to receive notification of arriving messages. This is a simple feature that the example service can add without too much difficulty.

To provide notifications, the service needs an interface for the listener objects. The code in Listing 10.11 provides a simple interface that the listener objects can implement to receive notification of a newly arrived `ExampleResponseMsg`.

Listing 10.11 Source Code for *ExampleServiceListener.java*

```
package com.newriders.jxta.chapter10;

/**
 * An interface to encapsulate an object that listens for notification
 * from the ExampleService of newly arrived ExampleResponseMsg messages.
 */
public interface ExampleServiceListener
{
    /**
     * Process the newly arrived ExampleResponseMsg message.
     *
     * @param   answer the object encapsulating the notification event.
     */
    public void processAnswer(ExampleServiceEvent answer);
}
```

The `ExampleListener` interface defines only a single method, `processAnswer`, that the service calls to notify the listener. Although it is perhaps a bit unnecessary for this simple service, the `processAnswer` method takes an instance of the `ExampleServiceEvent` class, shown in Listing 10.12, as a parameter.

Listing 10.12 Source Code for *ExampleServiceEvent.java*

```
package com.newriders.jxta.chapter10;

import java.util.EventObject;



/**
 * An object to encapsulate the event signaling the arrival of a
 * new ExampleResponseMsg at the ExampleService.
 */
public class ExampleServiceEvent extends EventObject
{
    /**
      * The response object that triggered this event.
      */
    private ExampleResponseMsg response = null;



    /**
     * Creates a new event object from the given source and
     * message object.
     *
     * @param   source the ExampleService source of the event.
     * @param   response the newly arrived ExampleResponseMsg message.
     */
    public ExampleServiceEvent(Object source, ExampleResponseMsg
response)
    {
        super(source);
        this.response = response;
```

```
    }

    /**
     * Returns the response that triggered this event.
     *
     * @return  the newly arrived response message.
     */
    public ExampleResponseMsg getResponse()
    {
        return response;
    }
}
```

The `ExampleServiceEvent` serves only to wrap the arriving `ExampleResponseMsg`, which is perhaps overkill for such a simple service. But in a more elaborate service, the event object could provide other valuable information. For example, a more sophisticated service might require information about the exact time of the message's arrival, the endpoint protocol implementation used to receive the message, or the source of the response. All of this information, which is not a part of the message contents, could be added to the event object and thereby provided to the listener in addition to the received message itself.

Creating the Example Service Interface

Although the `Service` interface could be directly implemented by the example service's implementation class, it is better to separate the definition of the service from its implementation. By defining an interface for the example service, a third-party developer can define a different implementation that can be used transparently.

For the example service, only three pieces of functionality are required:

- The capability to register an `ExampleServiceListener` object with the service, allowing the listener to be notified of incoming messages
- The capability to send an `ExampleQueryMsg` to peers in the group without formulating the `ExampleQueryMsg` manually

- The capability to unregister an `ExampleServiceListener` object from the service, thus preventing the listener from receiving further message arrival notifications

The `ExampleService` interface shown in Listing 10.13 provides all of this functionality in its methods.

Listing 10.13 Source Code for *ExampleService.java*

```
package com.newriders.jxta.chapter10;

import net.jxta.service.Service;


/**
 * An interface for the ExampleService. This interface defines the
 * operations that a developer can expect to use to manipulate the
 * ExampleService regardless of which underlying implementation of
 * the service is being used.
 */
public interface ExampleService extends Service
{
    /**
     * Add a listener object to the service. When new ExampleResponseMsg
     * responses arrive, the service will notify each registered listener.
     *
     * @param   listener the listener object to register with the service.
     */
    public void addListener(ExampleServiceListener listener);

    /**
     * Send a query to the network to determine the value of the given
     * base raised to the given power.
     *
     * @param   base the base for the exponentiation operation.
     * @param   power the exponent for the exponentiation operation.
     */
```

```
    public void findAnswer(double base, double power);

    /**
     * Remove a given listener object from the service. Once removed,
     * a listener will no longer be notified when a new ExampleResponseMsg
     * response arrives.
     *
     * @param   listener the listener object to unregister.
     */
    public void removeListener(ExampleServiceListener listener);
}
```

Notice that the interface doesn't implement the `Service` interface. This responsibility is left to the implementation of the `ExampleService` interface.

The *ExampleService* Implementation

The `ExampleService` implementation shown in Listing 10.14 provides the actual functionality provided by the service. It is responsible for registering with the Resolver service to accept queries from peers and managing the set of registered listeners.

Listing 10.14 Source Code for *ExampleServiceImpl.java*

```
package com.newriders.jxta.chapter10;

import java.io.ByteArrayInputStream;
import java.io.IOException;

import java.util.Vector;

import net.jxta.document.Advertisement;

import net.jxta.exception.PeerGroupException;
import net.jxta.exception.NoResponseException;
import net.jxta.exception.DiscardQueryException;
import net.jxta.exception.ResendQueryException;
```

```
import net.jxta.id.ID;

import net.jxta.impl.protocol.ResolverQuery;
import net.jxta.impl.protocol.ResolverResponse;

import net.jxta.peergroup.PeerGroup;

import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.protocol.ResolverQueryMsg;
import net.jxta.protocol.ResolverResponseMsg;

import net.jxta.resolver.QueryHandler;
import net.jxta.resolver.ResolverService;

import net.jxta.service.Service;


/**
 * The implementation of the ExampleService interface. This service
 * builds on top of the Resolver service to provide the query
 * functionality.
 */
public class ExampleServiceImpl implements ExampleService, QueryHandler
{
    /**
     * The Module Implementation advertisement for this service.
     */
    private Advertisement implAdvertisement = null;

    /**
     * The handler name used to register the Resolver handler.
     */
    private String handlerName = null;

    /**
```

```
 * The set of listener objects registered with the service.
 */
private Vector registeredListeners = new Vector();

/**
 * The peer group to which the service belongs.
 */
private PeerGroup myPeerGroup = null;

/**
 * The Resolver service used to send response messages.
 */
private ResolverService resolver = null;

/**
 * A unique query ID that can be used to track a query.
 */
private int queryID = 0;



/**
 * Add a listener object to the service. When new ExampleResponseMsg
 * responses arrive, the service will notify each registered listener.
 * This method is synchronized to prevent multiple threads from
 * altering the set of registered listeners simultaneously.
 *
 * @param   listener the listener object to register with the service.
 */
public synchronized void addListener(ExampleServiceListener listener)
{
    registeredListeners.addElement(listener);
}

/**
 * Send a query to the network to determine the value of the given
 * base raised to the given power.
```

```
 *
 * @param   base the base for the exponentiation operation.
 * @param   power the exponent for the exponentiation operation.
 */
public void findAnswer(double base, double power)
{
    // Make sure the service has been started.
    if (resolver != null)
    {
        // Create the query object using the given base and power.
        ExampleQueryMsg equery = new ExampleQueryMsg(base, power);
        String localPeerId = myPeerGroup.getPeerID().toString();

        // Wrap the query in a Resolver Query Message.
        ResolverQuery query = new ResolverQuery(handlerName,
            "JXTACRED", localPeerId, equery.toString(), queryID++);

        // Send the query using the Resolver service.
            resolver.sendQuery(null, query);
    }
}

/**
 * Returns the advertisement for this service. In this case, this is
 * the ModuleImplAdvertisement passed in when the service was
 * initialized.
 *
 * @return  the advertisement describing this service.
 */
public Advertisement getImplAdvertisement()
{
    return implAdvertisement;
}

/**
 * Returns an interface used to protect this service.
```

```
     *
     * @return  the wrapper object to use to manipulate this service.
     */
    public Service getInterface()
    {
        // We don't really need to provide an interface object to protect
        // this service, so this method simply returns the service itself.
        return this;
    }

    /**
     * Initialize the service.
     *
     * @param        group the PeerGroup containing this service.
     * @param        assignedID the identifier for this service.
     * @param        implAdv the advertisement specifying this service.
     * @exception    PeerGroupException is not thrown ever by this
     *               implementation.
     */
    public void init(PeerGroup group, ID assignedID, Advertisement implAdv)
        throws PeerGroupException
    {
        // Save the module's implementation advertisement.
        implAdvertisement = (ModuleImplAdvertisement) implAdv;

        // Use the assigned ID as the Resolver handler name.
        handlerName = assignedID.toString();

        // Save a reference to the group of which that this service
        // is a part.
        myPeerGroup = group;
    }

    /**
     * Processes the Resolver query message and returns a response.
     *
```

```
     * @param       query the message to be processed.
     * @exception   IOException if the query can't be read.
     */
    public ResolverResponseMsg processQuery(ResolverQueryMsg query)
        throws IOException, NoResponseException, DiscardQueryException,
            ResendQueryException
    {
        ResolverResponse response;
        ExampleQueryMsg eq;
        double answer = 0.0;

        try
        {
            // Extract the query message.
            eq = new ExampleQueryMsg(
                new
ByteArrayInputStream((query.getQuery()).getBytes()));
        }
        catch (Exception e)
        {
            throw new IOException();
        }

        // Perform the calculation.
        answer = Math.pow(eq.getBase(), eq.getPower());

        // Create the response message.
        ExampleResponseMsg er = new ExampleResponseMsg(
            eq.getBase(), eq.getPower(), answer);

        // Wrap the response message in a Resolver Response Message.
        response = new ResolverResponse(handlerName, "JXTACRED",
            query.getQueryId(), er.toString());

        // Return the message so that the Resolver service can handle
        // sending it.
```

```
    return response;
}

/**
 * Process a Resolver Response Message.
 *
 * @param   response a response message to be processed.
 */
public void processResponse(ResolverResponseMsg response)
{
    ExampleResponseMsg er;
    ExampleServiceEvent event;


    try
    {
        // Extract the message from the Resolver response.
        er = new ExampleResponseMsg(
            new ByteArrayInputStream(
                (response.getResponse()).getBytes()));

        // Create an event to send to the listeners.
        event = new ExampleServiceEvent(this, er);

        // Notify each of the registered listeners.
        if (registeredListeners.size() > 0)
        {
            ExampleServiceListener listener = null;

            for (int i = 0; i < registeredListeners.size(); i++)
            {
                listener = (ExampleServiceListener)
                    registeredListeners.elementAt(i);
                listener.processAnswer(event);
            }
        }
```

```
        }
        catch (Exception e)
        {
            // This is not the right type of response message, or
            // the message is improperly formed. Ignore the exception;
            // do nothing with the message.
        }
    }

    /**
     * Remove a given listener object from the service. Once removed,
     * a listener will no longer be notified when a new ExampleResponseMsg
     * response arrives.
     *
     * @param   listener the listener object to unregister.
     */
    public synchronized void removeListener(ExampleServiceListener
listener)
    {
        registeredListeners.removeElement(listener);
    }

    /**
     * Start the service.
     *
     * @param   args the arguments to the service. Not used.
     * @return  0 to indicate the service started.
     */
    public int startApp(String[] args)
    {
        // Now that the service is being started, set the ResolverService
        // object to use to handle queries and responses.
        resolver = myPeerGroup.getResolverService();
        // Add ourselves as a listener using the unique constructed
        // handler name.
        resolver.registerHandler(handlerName, this);
```

```
        return 0;
    }

    /**
     * Stop the service.
     */
    public void stopApp()
    {
        // Unregister ourselves as a listener.
        if (resolver != null)
        {
            resolver.unregisterHandler(handlerName);
        }
    }
}
```

Note

The service implementation must have a zero-argument constructor to allow the platform to load the service properly when initializing a peer group configured to use the service implementation. In the `ExampleServiceImpl` class, no constructor is defined, so the Java compiler generates a zero-argument constructor by default.

The `init` method implementation simply stores the passed parameters for later use. The passed ID will be used later to register the service with the Resolver service, and the given peer group will be used to obtain access to the Resolver service. Although nothing prevents the `ExampleServiceImpl` from registering with the Resolver in the `init` method, that task is performed in the `startApp` method. The `init` method is called to prepare the service, but the service shouldn't begin handling queries until the `startApp` method is called. Hence, the service doesn't register with the Resolver service until `startApp` is called. Conversely, the `stopApp` method unregisters the service with the Resolver service to prevent the service from handling queries when it has been stopped.

Adding the *ExampleService* Implementation to a Peer Group

The most difficult part of creating a new service is not creating the `Service` implementation, but adding it to a peer group. Adding the service requires the creation of the Module Class, Module Specification, and Module Implementation Advertisements describing the `Service` implementation. Usually it is preferred that the Module Class and Specification IDs be created beforehand so that their values are known for future reference. The simple application in Listing 10.15 generates the various required IDs and prints their values to the screen.

Listing 10.15 Source Code for *GenerateID.java*

```
package com.newriders.jxta.chapter10;

import net.jxta.id.IDFactory;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupID;

import net.jxta.platform.ModuleClassID;
import net.jxta.platform.ModuleSpecID;



/**
 * A simple application to generate a Module Class ID, Module Specification
 * ID, Peer Group ID, and Module Specification ID based on the standard
 * peer group Module Class ID.
 */
public class GenerateID
{
    /**
     * Generates the IDs.
     *
     * @param   args the command-line arguments. Ignored by this app.
     */
```

```
    public static void main(String[] args)
    {
        // Create an entirely new Module Class ID.
        ModuleClassID classID = IDFactory.newModuleClassID();

        // Create a Module Specification ID based on the generated
        // Module Class ID.
        ModuleSpecID specID = IDFactory.newModuleSpecID(classID);
        // Create an entirely new Peer Group ID.
        PeerGroupID groupID = IDFactory.newPeerGroupID();

        // Create a Module Specification ID based on the peer group
        // Module Class ID.
        ModuleSpecID groupSpecID = IDFactory.newModuleSpecID(
            PeerGroup.allPurposePeerGroupSpecID.getBaseClass());

        // Print out the generated IDs.
        System.out.println("Module Class ID: " + classID.toString());
        System.out.println("Module Spec ID: " + specID.toString());
        System.out.println("Peer Group ID: " + groupID.toString());
        System.out.println("Peer Group Module Spec ID: "
            + groupSpecID.toString());
    }
}
```

Although it is not essential, the `GenerateID` application also generates the Peer Group ID that will be used to create a peer group in the example. Creating a new peer group is a required part of adding a new service because the definition of the services offered by a peer group cannot change. The Module Implementation Advertisement associated with a peer group specifies which services the peer group offers. Because this advertisement is most likely cached throughout the network, allowing it to be changed would result in inconsistencies in the services offered by members of the peer group across the network. For the same reason that a Module Implementation Advertisement cannot be changed, a Peer Group Advertisement also cannot be altered. Therefore, offering a new service requires the creation of both a new Module Implementation Advertisement for the peer group and a new peer group that uses that Module Implementation Advertisement.

Because the example will have to create a new Module Implementation Advertisement, `GenerateID` creates the Module Specification ID that will be used for the new peer group's Module Implementation Advertisement. This ID is created using the Module Class ID of the standard peer group reference implementation.

With all those IDs generated, all that remains is to write an application such as Listing 10.16 that creates a new peer group that uses the new service.

Listing 10.16 Source Code for *ExampleServiceTest.java*

```
package com.newriders.jxta.chapter10;

import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.FlowLayout;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;

import javax.swing.JButton;
import javax.swing.JFrame;
import javax.swing.JLabel;
import javax.swing.JOptionPane;
import javax.swing.JPanel;
import javax.swing.JTextField;

import java.net.MalformedURLException;
import java.net.UnknownServiceException;
import java.net.URL;

import java.util.Enumeration;
import java.util.Hashtable;

import net.jxta.discovery.DiscoveryService;

import net.jxta.document.AdvertisementFactory;
```

```
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;

import net.jxta.exception.PeerGroupException;
import net.jxta.exception.ServiceNotFoundException;

import net.jxta.id.IDFactory;

import net.jxta.impl.peergroup.StdPeerGroupParamAdv;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.peergroup.PeerGroupFactory;

import net.jxta.platform.ModuleClassID;
import net.jxta.platform.ModuleSpecID;

import net.jxta.protocol.ModuleClassAdvertisement;
import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.protocol.ModuleSpecAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;


/**
 * An application to create a peer group, configure a new service for
 * the peer group, and then interact with other peers using that new
 * service.
 */
public class ExampleServiceTest implements ExampleServiceListener
{
    /**
     * The Module Class ID to use for the service.
     * YOU SHOULD REPLACE THIS WITH ONE YOU GENERATE
```

```
     * YOURSELF USING THE GenerateID APPLICATION!
     */
    private static final String refModuleClassID =
        "urn:jxta:uuid-128E938121DD4957B74B90EE27FDC61F05";

    /**
     * The Module Specification ID to use for the service.
     * YOU SHOULD REPLACE THIS WITH ONE YOU GENERATE
     * YOURSELF USING THE GenerateID APPLICATION!
     */
    private static final String refModuleSpecID =
        "urn:jxta:uuid-128E938121DD4957B74B90EE27FDC61FA385BCB"
        + "1BA504B0FA69F99FE84CDC25B06";

    /**
     * The Peer Group ID to use for the application.
     * YOU SHOULD REPLACE THIS WITH ONE YOU GENERATE
     * YOURSELF USING THE GenerateID APPLICATION!
     */
    private static final String refPeerGroupID =
        "urn:jxta:uuid-A87A7DD0762F47E88B2FB5452D47B3A802";

    /**
     * The peer group Module Specification ID to use for the application.
     * YOU SHOULD REPLACE THIS WITH ONE YOU GENERATE
     * YOURSELF USING THE GenerateID APPLICATION!
     */
    private static final String refPeerGroupSpec =
        "urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE00000001CB18295"
        + "F0DE94F99983AA2F00C1DE42F06";

    /**
     * The Net Peer Group for the application.
     */
    private PeerGroup netPeerGroup = null;
```

```
/**
 * The frame for the application user interface.
 */
private JFrame clientFrame = new JFrame("Exponentiator");

/**
 * The textfield for accepting the base input for the
 * exponentiation operation.
 */
private JTextField baseText = new JTextField(5);

/**
 * The textfield for accepting the power input for the
 * exponentiation operation.
 */
private JTextField powerText = new JTextField(5);

/**
 * The new group created by the application.
 */
private PeerGroup newGroup = null;
/**
 * Create the Module Class Advertisement for the service, using the
 * preconfigured ID in refModuleClassID.
 *
 * @return      the generated Module Class Advertisement.
 * @exception   UnknownServiceException, MalformedURLException thrown
 *              if the refModuleClassID is invalid or malformed.
 */
private ModuleClassAdvertisement createModuleClassAdv()
    throws UnknownServiceException, MalformedURLException
{
    // Create the class ID from the refModuleClassID string.
    ModuleClassID classID = (ModuleClassID) IDFactory.fromURL(
        new URL((refModuleClassID)));
```

```
    // Create the Module Class Advertisement.
    ModuleClassAdvertisement moduleClassAdv =
        (ModuleClassAdvertisement)
            AdvertisementFactory.newAdvertisement(
                ModuleClassAdvertisement.getAdvertisementType());

    // Configure the Module Class Advertisement.
    moduleClassAdv.setDescription(
        "A service to handle exponentiation math problems.");
    moduleClassAdv.setModuleClassID(classID);
    moduleClassAdv.setName("Exponentiator Class");

    // Return the advertisement to the caller.
    return moduleClassAdv;
}

/**
 * Create the Module Implementation Advertisement for the service,
 * using the specification ID in the passed in ModuleSpecAdvertisement
 * advertisement. Use the given ModuleImplAdvertisement to create the
 * compatibility element of the module impl specification.
 *
 * @param       groupImpl the ModuleImplAdvertisement of the parent
 *              peer group.
 * @param       moduleSpecAdv the source of the specification ID.
 * @return      the generated Module Implementation Advertisement.
 */
private ModuleImplAdvertisement createModuleImplAdv(
    ModuleImplAdvertisement groupImpl,
    ModuleSpecAdvertisement moduleSpecAdv)
{
    // Get the specification ID from the passed advertisement.
    ModuleSpecID specID = moduleSpecAdv.getModuleSpecID();

    // Create the Module Implementation Advertisement.
    ModuleImplAdvertisement moduleImplAdv = (ModuleImplAdvertisement)
```

```
            AdvertisementFactory.newAdvertisement(
                ModuleImplAdvertisement.getAdvertisementType());

        // Configure the Module Implementation Advertisement.
        moduleImplAdv.setCode(
            "com.newriders.jxta.chapter10.ExampleServiceImpl");
        moduleImplAdv.setCompat(groupImpl.getCompat());
        moduleImplAdv.setDescription(
            "Reference Exponentiator implementation");
        moduleImplAdv.setModuleSpecID(specID);
        moduleImplAdv.setProvider("Brendon J. Wilson");

        // Return the advertisement to the caller.
        return moduleImplAdv;
    }

    /**
     * Create the Module Specification Advertisement for the service,
     * using the preconfigured ID in refModuleSpecID.
     *
     * @return      the generated Module Class Advertisement.
     * @exception   UnknownServiceException, MalformedURLException thrown
     *              if the refModuleSpecID is invalid or malformed.
     */
    private ModuleSpecAdvertisement createModuleSpecAdv()
        throws UnknownServiceException, MalformedURLException
    {
        // Create the specification ID from the refModuleSpecID string.
        ModuleSpecID specID = (ModuleSpecID) IDFactory.fromURL(
            new URL((refModuleSpecID)));

        // Create the Module Specification Advertisement.
        ModuleSpecAdvertisement moduleSpecAdv =
(ModuleSpecAdvertisement)
            AdvertisementFactory.newAdvertisement(
                ModuleSpecAdvertisement.getAdvertisementType());
```

```
    // Configure the Module Specification Advertisement.
    moduleSpecAdv.setCreator("Brendon J. Wilson");
    moduleSpecAdv.setDescription(
        "A specification for an exponentiation service.");
    moduleSpecAdv.setModuleSpecID(specID);
    moduleSpecAdv.setName("Exponentiator Spec");
    moduleSpecAdv.setSpecURI(
        "http://www.brendonwilson.com/projects/jxta");
    moduleSpecAdv.setVersion("1.0");

    // Return the advertisement to the caller.
    return moduleSpecAdv;
}

/**
 * Creates a peer group and configures the ExampleService
 * implementation to run as a peer group service.
 *
 * @exception  Exception, PeerGroupException if there is a problem
 *             while creating the peer group or the service
 *             advertisements.
 */
public void createPeerGroup() throws Exception, PeerGroupException
{
    // The name and description for the peer group.
    String name = "CreatePeerGroup";
    String description =
        "An example peer group to test peer group creation";

    // The Discovery service to use to publish the module and peer
    // group advertisements.
    DiscoveryService discovery = netPeerGroup.getDiscoveryService();

    // Obtain a preformed ModuleImplAdvertisement to use when creating
    // the new peer group. This is the Module Implementation
```

```
// Advertisement of the Net Peer Group and contains all of the
// services and applications already configured to run in that peer
// group. Using this method simplifies the task of creating a new
// peer group and configuring a new service.
ModuleImplAdvertisement implAdv =
    netPeerGroup.getAllPurposePeerGroupImplAdvertisement();

// Create the Module Class Advertisement.
ModuleClassAdvertisement moduleClassAdv = createModuleClassAdv();

// Create the Module Specification Advertisement.
ModuleSpecAdvertisement moduleSpecAdv = createModuleSpecAdv();

// Create the Module Implementation Advertisement.
ModuleImplAdvertisement moduleImplAdv =
    createModuleImplAdv(implAdv,moduleSpecAdv);

// Get the parameters for the peer group's Module Implementation
// Advertisement to add our service.
StdPeerGroupParamAdv params =
    new StdPeerGroupParamAdv(implAdv.getParam());

// Get the services from the parameters.
Hashtable services = params.getServices();

// Add our service to the set of services.
services.put(moduleClassAdv.getModuleClassID(), moduleImplAdv);

// Set the services on the parameters, and set the parameters on
// the implementation advertisement.
params.setServices(services);
implAdv.setParam((StructuredDocument) params.getDocument(
    new MimeMediaType("text", "xml")));

// VERY IMPORTANT! You must change the Module Specification ID
// for the implementation advertisement. If you don't, the new
```

```
        // peer group's Module Specification ID will still point to the
        // old specification, and the new service will not be loaded.
        implAdv.setModuleSpecID((ModuleSpecID) IDFactory.fromURL(
            new URL(refPeerGroupSpec)));

        // Publish the Module Class and Specification Advertisements.
        discovery.publish(moduleClassAdv, DiscoveryService.ADV);
        discovery.remotePublish(moduleClassAdv, DiscoveryService.ADV);
        discovery.publish(moduleSpecAdv, DiscoveryService.ADV);
        discovery.remotePublish(moduleSpecAdv, DiscoveryService.ADV);
        discovery.publish(implAdv, DiscoveryService.ADV);
        discovery.remotePublish(implAdv, DiscoveryService.ADV);

        // Create the Peer Group ID.
        PeerGroupID groupID = (PeerGroupID) IDFactory.fromURL(
            new URL((refPeerGroupID)));

        // Create the new group using the group ID, advertisement, name,
        // and description.
        newGroup = netPeerGroup.newGroup(groupID, implAdv, name,
            description);

        // Need to publish the group remotely only because newGroup()
        // handles publishing to the local peer.
        PeerGroupAdvertisement groupAdv =
            newGroup.getPeerGroupAdvertisement();
        discovery.remotePublish(groupAdv, DiscoveryService.GROUP);
    }

    /**
     * Starts the JXTA platform.
     *
     * @exception   PeerGroupException thrown if the platform can't
     *              be started.
     */
    public void initializeJXTA() throws PeerGroupException
```

```
{
    netPeerGroup = PeerGroupFactory.newNetPeerGroup();
}

/**
 * Starts the application.
 *
 * @param   args the command-line arguments passed to the application.
 */
public static void main(String[] args)
{
    ExampleServiceTest test = new ExampleServiceTest();
    try
    {
        // Initialize the JXTA platform.
        test.initializeJXTA();

        // Create the group.
        test.createPeerGroup();

        // Show a GUI to accept input.
        test.showGUI();
    }
    catch (Exception e)
    {
        System.out.println("Error starting JXTA platform: " + e);
        System.exit(1);
    }
}

/**
 * The implementation of the ExampleServiceListener interface. This
 * allows us to display a message each time a message is received by
 * the ExampleService.
 *
 * @param   answer the event containing the newly arrived message.
```

```
 */
public void processAnswer(ExampleServiceEvent event)
{
    // Extract the response message from the event object.
    ExampleResponseMsg er = event.getResponse();

    // Print out the answer given in the response.
    String answer = "The value of " + er.getBase() + " raised to "
        + er.getPower() + " is: " + er.getAnswer();
    JOptionPane.showMessageDialog(null, answer, "Answer Received!",
        JOptionPane.INFORMATION_MESSAGE);
}

/**
 * Convenience method to find the service and use it to send
 * a query to other peers' ExampleService.
 *
 * @param   base the base for the exponentiation query.
 * @param   power the power for the exponentiation query.
 */
private void sendMessage(String base, String power)
{
    try
    {
        // Convert the input to numbers.
        double baseValue = Double.parseDouble(base);
        double powerValue = Double.parseDouble(power);

        // Find the service on the peer group.
        ModuleClassID classID = (ModuleClassID) IDFactory.fromURL(
            new URL((refModuleClassID)));
        ExampleService exponentiator =
            (ExampleService) newGroup.lookupService(classID);
        exponentiator.findAnswer(baseValue, powerValue);
    }
    catch (NumberFormatException e)
```

```
        {
            // Warn the user.
            JOptionPane.showMessageDialog(null, "The base and power must
"
                + "both be numbers!", "Input Error!",
                JOptionPane.ERROR_MESSAGE);
        }
        catch (Exception e2)
        {
            // Warn the user.
            JOptionPane.showMessageDialog(null, "Error finding service!",
                "Service Error!", JOptionPane.ERROR_MESSAGE);
        }
    }

    /**
     * Displays a user interface to allow the user to send queries to
     * other peers.
     *
     * @exception   exceptions thrown only if the new service can't be
     *              found.
     */
    private void showGUI() throws UnknownServiceException,
        MalformedURLException, ServiceNotFoundException
    {
        JButton sendButton = new JButton("Send Message");
        JButton quitButton = new JButton("Quit");
        JPanel sendPane = new JPanel();
        JLabel baseLabel = new JLabel("Base:");
        JLabel powerLabel = new JLabel("Power:");
        Container pane = clientFrame.getContentPane();


        // Populate the GUI frame.
        sendPane.setLayout(new FlowLayout());
        sendPane.add(baseLabel);
```

```
        sendPane.add(baseText);
        sendPane.add(powerLabel);
        sendPane.add(powerText);
        sendPane.add(sendButton);
        sendPane.add(quitButton);
        pane.setLayout(new BorderLayout());
        pane.add(sendPane, BorderLayout.SOUTH);

        // Set up listeners for the buttons.
        sendButton.addActionListener(
            new ActionListener() {
                public void actionPerformed(ActionEvent e)
                {
                    // Send the message.
                    sendMessage(baseText.getText(),
powerText.getText());

                    // Clear the text.
                    baseText.setText("");
                    powerText.setText("");
                }
            }
        );

        quitButton.addActionListener(
            new ActionListener()
            {
                public void actionPerformed(ActionEvent e)
                {
                    clientFrame.hide();
                    // Stop the JXTA platform. Currently, there isn't any
                    // nice way to do this.
                    System.exit(0);
                }
            }
        );
```

```
        // Find the new service on the peer group and add ourselves
        // as a listener.
        ModuleClassID classID = (ModuleClassID) IDFactory.fromURL(
            new URL((refModuleClassID)));
        ExampleService exponentiator =
            (ExampleService) newGroup.lookupService(classID);
        exponentiator.addListener(this);

        // Pack and display the user interface.
        clientFrame.pack();
        clientFrame.show();
    }
}
```

To simplify the task of creating a Module Implementation Advertisement for a new peer group, a developer can use the `getAllPurposePeerGroupImplAdvertisement` on an existing peer group. This method provides a copy of the peer group's `ModuleImplAdvertisement` containing the parameters that specify the services offered by the peer group. In the reference implementation, these parameters can be manipulated via the `StdPeerGroupParamAdv` class provided in the `net.jxta.impl.peergroup` package.

The process of creating a new peer group with a new service can be con-fusing, so here are the essential steps that the `ExampleServiceTest` executes:

- **`createModuleClassAdv`—** This method creates the Module Class Advertisement for the example service, using a Module Class ID hard-coded in the `ExampleServiceTest`. This Module Class ID was generated using `GenerateID`. The Module Class Advertisement is configured with the Module Class ID, plus a simple name and description.

- **`createModuleSpecAdv`—** This method creates the Module Specification Advertisement for the example service, using a Module Specification ID hard-coded in the `ExampleServiceTest`. This Module Specification ID was generated using `GenerateID`. The Module Specification Advertisement is configured to provide version information on the new service and where to find a document describing the specification of the service.

- **`createModuleImplAdv`**— This method creates the Module Implementation Advertisement for the example service, using the same Module Specification ID used when creating the Module Specification Advertisement. The Module Implementation Advertisement is configured to provide information on the implementation of the service. This is where the `ExampleServiceImpl` code is bound to a Module Implementation Advertisement. To provide the same compatibility as other services on the peer, the generic implementation advertisement retrieved using the `getAllPurposePeerGroupImplAdvertisement` method is passed to this method. This advertisement is used as the source of the compatibility information configured on the Module Implementation Advertisement for the new service.

After the various module advertisements for the example service have been created, the Module Implementation Advertisement for the new service must be added to the Module Implementation Advertisement for the peer group. The `ExampleServiceTest` application performs the following steps to alter the generic peer group Module Implementation Advertisement returned by the `getAllPurposePeerGroupImplAdvertisement` method:

1. Extract the parameters from the generic peer group Module Implementation Advertisement using `getParam`, and create a `StdPeerGroupParams` object. This object deals with the format for the parameters used by the reference implementation of the `PeerGroup` interface.

2. Extract the parameter's `Hashtable` of services using `getServices`.

3. Add the new service implementation advertisement using the `put` method. In the reference implementation, services are added to the `Hashtable` using the service's class ID as a key.

4. Set the service `Hashtable` on the parameters using `setServices`.

5. Set the parameters on the peer group's Module Implementation Advertisement using `setParam`.

6. Change the Module Implementation Advertisement's Module Specification ID. This is a very important step! If the implementation's Module Implementation's Module Specification ID isn't changed, the new peer group will use the Module

Specification ID of the peer group that provided the generic peer group Module Implementation Advertisement. When the new peer group is created, the platform will search for an implementation of the old module specification; therefore, the new service will never be loaded.

When those steps are completed, the new peer group can be created using the new peer group Module Implementation Advertisement. The new peer group's Module Implementation Advertisement causes the new service to be loaded and to start the new service.

Running the *ExampleServiceTest* Application

To run the `ExampleServiceTest` and see the example service in action, follow these steps:

1. Compile all the source code.
2. Place the resulting class files in a new directory.
3. Copy all the JXTA JAR files into this new directory.
4. Create a copy of this directory.
5. Start the `ExampleServiceTest` from the first directory by opening a command console, changing to the first directory, and executing this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar com.newriders.jxta.chapter10.ExampleServiceTest
```

The user interface for the application should appear. Start the `ExampleServiceTest` application from the second directory in the same way. After the user interface appears, you should be able to use the user interface to send a message via the example service between the two applications.

Summary

In this chapter, you've seen peer groups and peer group services and learned how they are related. As part of this discussion, this chapter explored how modules can be used within JXTA and how JXTA provides support for multiple versions and implementations of a module. Finally, the chapter demonstrated how to create a new service module and add it to a new peer group. In the next chapter, all the elements of the previous chapters are brought together in a sample application to demonstrate the power of JXTA.

Part III: Putting It All Together

Chapter 11. A Complete Sample Application

At this point in the book, you should be familiar with all the pieces involved in a JXTA solution, but not necessarily how to put them together into a complete application. This chapter provides a complete sample application that illustrates how to assemble the pieces provided by the JXTA reference implementation into a complete P2P solution.

The sample application in this chapter demonstrates the creation of a JXTA-based chat application with simple presence management. The application is similar to other popular instant-messaging clients, but it incorporates fewer features. The sample application incorporates many of the JXTA protocols to provide a complete solution.

The main features of this application are the capability to chat with a remote user and monitor remote users' presence status. This status allows a user to determine whether one of his contacts (or "buddies") is currently online, offline, busy, or temporarily away from the computer.

Creating the Presence Service

The Presence service provides a mechanism for exchanging presence information with another user. For this application, the Presence service is fairly unsophisticated and doesn't attempt to address more complex presence-management problems. For example, this Presence service assumes that a user is on the network at only a single location using a single peer. In addition, the Presence service assumes that JXTA provides a reliable transport, which is not always a good assumption. Although the reference implementation uses TCP, a reliable transport, there are no guarantees on a given JXTA peer that the peer will be using a reliable transport.

The JXTA Community is currently working on a fully featured Presence Management Framework that will provide much more functionality than this sample application's Presence service. For more information on the Presence Management Framework, see the project web site at `presence.jxta.org`. This example Presence service is simply an example designed to show how the various pieces explored over the course of this book fit together into a single application.

At first glance, it might appear that the Presence service should be built using the Resolver service. However, building the Presence service using the Resolver service would require a peer to send a query to a remote peer every time that it required presence information. The network overhead incurred by this technique would be undesirable.

It would be better if presence information could be published using the Discovery service, thus allowing presence information to be cached by other peers. Of course, the risk here is that the presence information might be stale, but this can be resolved by publishing the advertisement with a short lifetime. Another disadvantage is that this method does not scale well to large numbers of peers. This is something that is acceptable for this simple application, but it would not be acceptable in a large-scale P2P solution.

To implement the Presence service, three components are needed:

- **An advertisement—** The Presence service needs a format for the presence information to be exchanged with other peers using the Discovery service. The formatting and parsing logic for the advertisement must be implemented to allow the Presence service to handle the advertisement in an encapsulated fashion.

- **A service—** The Presence service itself needs to provide an interface that third-party developers can use to interact with the service. An implementation of the service's interface is required to handle the details of using the Discovery service to publish and discover presence information.

- **A listener—** This application needs some way of receiving notification that new presence information has been received by the Presence service. A listener interface that can be implemented and registered with the Presence service solves this problem.

The creation of these elements is the subject of the next two sections.

The Presence Advertisement

The Presence Advertisement is responsible for describing the current presence status of a particular user. To uniquely identify both a peer and the user, the Presence Advertisement needs the Peer ID, plus one other piece of identification that is unique to the user. For this purpose, the Presence Advertisement uses the user's email address to uniquely identify the user.

To represent the presence information, the Presence Advertisement uses the XML format shown in Listing 11.1.

Listing 11.1 The Presence Advertisement XML

```
<?xml version="1.0" encoding="UTF-8"?>
<PresenceAdvertisement>
  <PeerID> . . . </PeerID>
  <EmailAddress> . . . </EmailAddress>
  <PresenceStatus> . . . </PresenceStatus>
  <Name> . . . </Name>
</PresenceAdvertisement>
```

The content of the Presence Advertisement describes all the elements related to a user's presence on the P2P network:

- **`PeerID`**— A required element containing the Peer ID identifying the peer that the user is currently using on the network.
- **`EmailAddress`**— A required element containing the user's email address. The email address is used as a unique identifier for a user.
- **`PresenceStatus`**— A required element containing an integer representing the user's presence status. A value of `0` indicates that the user is offline, `1` indicates that the user is online, `2` indicates that the user is currently busy, and `3` indicates that the user is temporarily away from the computer.

- **`Name`**— An optional element containing a display name or common name for the user.

To implement the Presence Advertisement, you define an abstract class derived from the `net.jxta.document.Advertisement` class. This class, `PresenceAdvertisement`, is shown in Listing 11.2.

Listing 11.2 Source Code for *PresenceAdvertisement.java*

```
package com.newriders.jxta.chapter11.protocol;

import net.jxta.document.Advertisement;
import net.jxta.document.Document;
import net.jxta.document.MimeMediaType;

import net.jxta.id.ID;

import net.jxta.protocol.PipeAdvertisement;

import com.newriders.jxta.chapter11.presence.PresenceService;


/**
 * An abstract class defining an advertisement containing the elements used
 * to describe a user's presence status. A user is assumed to be uniquely
 * described by his or her email address.
 */
public abstract class PresenceAdvertisement extends Advertisement
{
    /**
     * The root element for the advertisement's XML document.
     */
    private static final String advertisementType =
"PresenceAdvertisement";

    /**
```

```
 * The email address identifying the user whose presence information
 * this advertisement describes.
 */
private String emailAddress = null;

/**
 * A simple name for the user specified by the advertisement's
 * email address.
 */
private String name = null;

/**
 * The Peer ID locating the peer on the network.
 */
private String peerID = null;
/**
 * A simple descriptor identifying the user's presence status.
 * The user can indicate that he or she is online, offline, busy, or
 * away.
 */
private int presenceStatus = PresenceService.OFFLINE;



/**
 * Returns the advertisement type for the advertisement's document.
 *
 * @return  the advertisement type String.
 */
public static String getAdvertisementType()
{
    return advertisementType;
}

/**
 * Returns the email address String describing the user whose presence
 * status is described by this advertisement.
```

```
     *
     * @return  the email address for the advertisement.
     */
    public String getEmailAddress()
    {
        return emailAddress;
    }

    /**
     * Returns a unique identifier for this document. There is none for
     * this advertisement type, so this method returns the null ID.
     *
     * @return  the null ID.
     */
    public ID getID()
    {
        return ID.nullID;
    }

    /**
     * Returns the simple name for the user described by this advertisement.
     *
     * @return  the user's name.
     */
    public String getName()
    {
        return name;
    }

    /**
     * Returns the Peer ID of the user described by this advertisement.
     *
     * @return  the Peer ID of the user.
     */
    public String getPeerID()
    {
```

```
    return peerID;
}

/**
 * Returns the presence status information of the user described by
 * this advertisement.
 *
 * @return  the user's status information.
 */
public int getPresenceStatus()
{
    return presenceStatus;
}

/**
 * Sets the email address String describing the user whose presence
 * status is described by this advertisement.
 *
 * @param   emailAddress the email address for the advertisement.
 */
public void setEmailAddress(String emailAddress)
{
    this.emailAddress = emailAddress;
}
/**
 * Sets the simple name for the user described by this advertisement.
 *
 * @param   name the user's name.
 */
public void setName(String name)
{
    this.name = name;
}

/**
 * Sets the Peer ID identifying the peer's location on the P2P network.
```

```
     *
     * @param   peerID the Peer ID for the advertisement.
     */
    public void setPeerID(String peerID)
    {
        this.peerID = peerID;
    }

    /**
     * Sets the presence status information of the user described by this
     * advertisement.
     *
     * @param   presenceStatus the user's status information.
     */
    public void setPresenceStatus(int presenceStatus)
    {
        this.presenceStatus = presenceStatus;
    }
}
```

The `PresenceAdvertisement` class defines basic accessors to set and retrieve the advertisement's various parameters. In addition, the class defines the static `getAdvertisementType` method to return the root element tag used by the Presence Advertisement.

`PresenceAdvertisement` also defines the `getID` method that is used by the Cache Manager to index the advertisement in the cache. The `ID` returned by `getID` should uniquely identify the advertisement. To avoid having to implement your own `ID` implementation, `PresenceAdvertisement` returns `ID.nullID`. This null `ID` prompts the Cache Manager to use a hash of the advertisement to index the advertisement in the cache, and it is sufficient for your purposes.

The `Advertisement.getDocument` method is not defined by

`PresenceAdvertisement`, to allow the implementation of `PresenceAdvertisement` to define logic for parsing and formatting a Presence Advertisement. This method is

implemented by the `PresenceAdv` subclass, shown in Listing 11.3, using the JXTA reference implementation.

Listing 11.3 Source Code for *PresenceAdv.java*

```
package com.newriders.jxta.chapter11.impl.protocol;

import java.io.InputStream;
import java.io.IOException;
import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.Advertisement;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.Element;
import net.jxta.document.Document;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredDocumentUtils;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;

import net.jxta.protocol.PipeAdvertisement;

import com.newriders.jxta.chapter11.protocol.PresenceAdvertisement;


/**
 * An implementation of the PresenceAdvertisement abstract class. This
 * class is responsible for parsing and formatting the XML document
 * used to define presence information for a peer.
 */
public class PresenceAdv extends PresenceAdvertisement
{
```

```
/**
 * A convenient constant for the XML MIME type.
 */
private static final String mimeType = "text/xml";
/**
 * The element name for the presence advertisement's email address info.
 */
private static final String tagEmailAddress = "EmailAddress";

/**
 * The element name for the presence advertisement's simple name info.
 */
private static final String tagName = "Name";

/**
 * The element name for the presence advertisement's Peer ID.
 */
private static final String tagPeerID = "PeerID";

/**
 * The element name for the presence advertisement's status info.
 */
private static final String tagPresenceStatus = "PresenceStatus";


/**
 * An Instantiator used by the AdvertisementFactory to instantiate
 * this class in an abstract fashion.
 */
public static class Instantiator
    implements AdvertisementFactory.Instantiator
{
    /**
     * Returns the identifying type of this advertisement.
     *
     * @return  the name of the advertisement's root element.
```

```
         */
        public String getAdvertisementType()
        {
            return PresenceAdvertisement.getAdvertisementType();
        }

        /**
         * Returns a new PresenceAdvertisement implementation instance.
         *
         * @return  a new presence advertisement instance.
         */
        public Advertisement newInstance()
        {
            return new PresenceAdv();
        }

        /**
         * Instantiates a new PresenceAdvertisement implementation
instance
         * populated from the given root element.
         *
         * @param   root the root of the object tree to use to populate the
         *          advertisement object.
         * @return  a new populated presence advertisement instance.
         */
        public Advertisement newInstance(Element root)
        {
            return new PresenceAdv(root);
        }
    };

    /**
     * Creates a new presence advertisement.
     */
    public PresenceAdv()
    {
```

```
    super();
}

/**
 * Creates a new presence advertisement by parsing the given stream.
 *
 * @param       stream the InputStream source of the advertisement data.
 * @exception   IOException if the advertisement can't be parsed from
 *              the stream.
 */
public PresenceAdv(InputStream stream) throws IOException
{
    super();

    StructuredTextDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            new MimeMediaType(mimeType), stream);
    readAdvertisement(document);
}

/**
 * Creates a new presence advertisement by parsing the given document.
 *
 * @param   document the source of the advertisement data.
 */
public PresenceAdv(Element document) throws IllegalArgumentException
{
    super();

    readAdvertisement((TextElement) document);
}

/**
 * Returns a Document object containing the advertisement's
 * document tree.
 *
```

```
     * @param        asMimeType the desired MIME type for the
     *               advertisement rendering.
     * @return       the Document containing the advertisement's document
     *               object tree.
     * @exception    IllegalArgumentException thrown if either the email
     *               address or the Peer ID is null.
     */
    public Document getDocument(MimeMediaType asMimeType)
        throws IllegalArgumentException
    {
        // Check that the required elements are present.
        if ((null != getEmailAddress()) && (null != getPeerID()))
        {
            PipeAdvertisement pipeAdv = null;

            StructuredDocument document = (StructuredTextDocument)
                StructuredDocumentFactory.newStructuredDocument(
                    asMimeType, getAdvertisementType());
            Element element;

            // Add the Peer ID information.
            element = document.createElement(tagPeerID, getPeerID());
            document.appendChild(element);
            // Add the email address information.
            element = document.createElement(
                tagEmailAddress, getEmailAddress());
            document.appendChild(element);

            // Add the display name information, if any.
            if (null != getName())
            {
                element = document.createElement(tagName, getName());
                document.appendChild(element);
            }

            // Add the presence status information.
```

```
            element = document.createElement(tagPresenceStatus,
                Integer.toString(getPresenceStatus()));
            document.appendChild(element);

            return document;
        }
        else
        {
            throw new IllegalArgumentException(
                "Missing email address or peer ID!");
        }
    }

    /**
     * Parses the given document tree for the presence advertisement.
     *
     * @param      document the object containing the presence
     *             advertisement data.
     * @exception   IllegalArgumentException if the document is not a
     *            presence advertisement, as expected.
     */
    public void readAdvertisement(TextElement document)
        throws IllegalArgumentException
    {
        if (document.getName().equals(getAdvertisementType()))
        {
            Enumeration elements = document.getChildren();
            while (elements.hasMoreElements())
            {
                TextElement element = (TextElement)
elements.nextElement();

                // Check for the email address element.
                if (element.getName().equals(tagEmailAddress))
                {
                    setEmailAddress(element.getTextValue());
```

```
                continue;
            }

            // Check for the display name element.
            if (element.getName().equals(tagName))
            {
                setName(element.getTextValue());
                continue;
            }

            // Check for the email address element.
            if (element.getName().equals(tagPresenceStatus))
            {
                setPresenceStatus(
                    Integer.parseInt(element.getTextValue()));
                continue;
            }

            // Check for the Peer ID element.
            if (element.getName().equals(tagPeerID))
            {
                setPeerID(element.getTextValue());
                continue;
            }
        }
    }
    else
    {
        throw new IllegalArgumentException(
            "Not a PresenceAdvertisement document!");
    }
}

/**
 * Returns an XML String representation of the advertisement.
 *
```

```
     * @return  the XML String representing this advertisement.
     */
    public String toString()
    {
        try
        {
            StringWriter out = new StringWriter();
            StructuredTextDocument doc = (StructuredTextDocument)
                getDocument(new MimeMediaType(mimeType));
            doc.sendToWriter(out);

            return out.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

In addition to providing a `getDocument` implementation, the `PresenceAdv` class provides several constructors that provide advertisement parsing functionality. All the parsing and formatting functionality is built using the `net.jxta.document` classes to handle manipulating the XML object tree.

In Chapter 4, "The Peer Discovery Protocol," you learned about the `AdvertisementFactory` class and how it could be used to instantiate an `Advertisement` implementation in an abstract manner using a `String`. Usually, this `String` comes from the abstract advertisement implementation class's `getAdvertisementType` method:

```
PeerAdvertisement advertisement =
    (PeerAdvertisement)
        AdvertisementFactory.newAdvertisement(
            PeerAdvertisement.getAdvertisementType());
```

For `AdvertisementFactory` to be capable of doing the same with the Presence Advertisement implementation, the implementation class must be registered with `AdvertisementFactory`. To register an implementation class, the application will need to call `AdvertisementFactory.registerAdvertisementInstance`:

```
public static boolean registerAdvertisementInstance(
    String rootType, Instantiator instantiator)
```

The `rootType String` defines the advertisement type `String` that will be mapped to the advertisement implementation. The `instantiator` parameter is an instance of an implementation of the `AdvertisementFactory.Instantiator` class. The `PresenceAdv` class shown in Listing 11.3 provides an implementation of this class that the `AdvertisementFactory` can use to create a new `PresenceAdv` instance. To register the `PresenceAdv` implementation with `AdvertisementFactory`, the sample application must execute the following:

```
AdvertisementFactory.registerAdvertisementInstance(
    PresenceAdvertisement.getAdvertisementType(),
    new PresenceAdv.Instantiator());
```

This call needs to be executed when the application starts, before any other class attempts to use `AdvertisementFactory` to instantiate a Presence Advertisement. In the example, `registerAdvertisementInstance` is called in the Presence service's `init` method to ensure that the advertisement type is registered.

The Presence Service Definition

Although this application could handle publishing and discovering Presence Advertisements directly, it would be better if a developer could avoid handling the Presence Advertisement and interacting with the Discovery service. If you define an interface for the Presence service, the solution will be more flexible and developers will be shielded from future implementation changes.

For example, if you didn't define an interface, a developer would have to use the Presence Advertisement classes and the Discovery service directly. What happens if the developer decides later that the application needs to use a mechanism other than the Discovery service for distributing and discovering Presence Advertisements? The application's code will probably need to be changed in several places. However, if the basic functionality of distributing and discovering presence information is defined as an interface, the developer can simply provide a different implementation of the interface and change the implementation that is used by the application.

The interface for the Presence service must allow a developer to do only two things: announce presence information and find presence information. Part of finding presence information involves notifying listener objects when presence information is found or received, necessitating some way of registering and unregistering listeners. All this functionality is defined by the `PresenceService` interface shown in Listing 11.4.

Listing 11.4 Source Code for *PresenceService.java*

```
package com.newriders.jxta.chapter11.presence;

import net.jxta.service.Service;


/**
 * An interface for the Presence service, a service that allows peers to
 * exchange presence status information specifying their current status
 * (offline, online, busy, away). This interface defines the operations
 * that a developer can expect to use to manipulate the Presence service,
 * regardless of which underlying implementation of the service is being
 * used.
 */
public interface PresenceService extends Service
{
    /**
     * The module class ID for the Presence class of service.
     */
    public static final String refModuleClassID =
```

```
    "urn:jxta:uuid-59A9A948905341119EAB8630EED42AB905";

/**
 * A status value indicating that a user is currently online but
 * is temporarily away from the device.
 */
public static final int AWAY= 3;

/**
 * A status value indicating that a user is currently online but
 * is busy and does not want to be disturbed.
 */
public static final int BUSY = 2;

/**
 * A status value indicating that a user is currently offline.
 */
public static final int OFFLINE = 0;

/**
 * A status value indicating that a user is currently online.
 */
public static final int ONLINE = 1;


/**
 * Add a listener object to the service. When a new Presence Response
 * Message arrives, the service will notify each registered listener.
 *
 * @param   listener the listener object to register with the service.
 */
public void addListener(PresenceListener listener);

/**
 * Announce updated presence information within the peer group.
 *
```

```
     * @param   presenceStatus the updated status for the user identified
     *          by the email address.
     * @param   emailAddress the email address used to identify the user
     *          associated with the presence info.
     * @param   name a display name for the user associated with the
     *          presence info.
     */
    public void announcePresence(int presenceStatus, String emailAddress,
        String name);

    /**
     * Sends a query to find presence information for the user specified
     * by the given email address. Any response received by the service
     * will be dispatched to registered PresenceListener objects.
     *
     * @param   emailAddress the email address to use to find presence
info.
     */
    public void findPresence(String emailAddress);

    /**
     * Removes a given listener object from the service. Once removed,
     * a listener will no longer be notified when a new Presence Response
     * Message arrives.
     *
     * @param   listener the listener object to unregister.
     */
    public boolean removeListener(PresenceListener listener);
}
```

To allow developers to handle notification of newly received presence information, the `PresenceService` class enables a developer to register and unregister a listener using the `addListener` and `removeListener` methods. The `PresenceListener` interface used by both of these methods is shown in Listing 11.5.

Listing 11.5 Source Code for *PresenceListener.java*

```
package com.newriders.jxta.chapter11.presence;

import com.newriders.jxta.chapter11.protocol.PresenceAdvertisement;


/**
 * An interface to encapsulate an object that listens for notification
 * from the PresenceService of newly arrived presence information.
 */
public interface PresenceListener
{
    /**
     * Notify the listener of newly arrived presence information.
     *
     * @param   presenceInfo the newly received presence information.
     */
    public void presenceUpdated(PresenceAdvertisement presenceInfo);
}
```

The implementation of `PresenceService` that will be used by the sample application relies on the `DiscoveryService` to handle publishing and discovering Presence Advertisements. The `PresenceService` implementation is shown in Listing 11.6.

Listing 11.6 Source Code for *PresenceServiceImpl.java*

```
package com.newriders.jxta.chapter11.impl.presence;

import java.io.ByteArrayInputStream;
import java.io.IOException;

import java.util.Enumeration;
import java.util.Vector;

import net.jxta.discovery.DiscoveryEvent;
```

```
import net.jxta.discovery.DiscoveryListener;
import net.jxta.discovery.DiscoveryService;

import net.jxta.document.Advertisement;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;

import net.jxta.exception.PeerGroupException;

import net.jxta.id.ID;

import net.jxta.impl.protocol.DiscoveryResponse;

import net.jxta.peergroup.PeerGroup;

import net.jxta.protocol.DiscoveryResponseMsg;
import net.jxta.protocol.ModuleImplAdvertisement;

import net.jxta.service.Service;

import com.newriders.jxta.chapter11.impl.protocol.PresenceAdv;

import com.newriders.jxta.chapter11.presence.PresenceListener;
import com.newriders.jxta.chapter11.presence.PresenceService;

import com.newriders.jxta.chapter11.protocol.PresenceAdvertisement;


/**
 * The implementation of the PresenceService interface. This service
 * builds on top of the Discovery service to provide the functionality
 * for requesting and providing presence information.
 */
public class PresenceServiceImpl implements PresenceService,
    DiscoveryListener
{
```

```
    /**
     * The Module Specification ID for the Presence service.
     */
    public static final String refModuleSpecID =
        "urn:jxta:uuid-59A9A948905341119EAB8630EED42AB"
        + "9F4611FF6377C4931AE71BE299B9F34DF06";

    /**
     * The default expiration timeout for published presence
advertisements.
     * Set to 1 minute.
     */
    private static final int DEFAULT_EXPIRATION = 1000 * 60 * 1;

    /**
     * The default lifetime for published presence advertisements.
     * Set to 1 minutes.
     */
    private static final int DEFAULT_LIFETIME = 1000 * 60 * 5;

    /**
     * The element name for the presence advertisement's email address info.
     */
    private static final String tagEmailAddress = "EmailAddress";

    /**
     * The Discovery service used to publish presence information.
     */
    private DiscoveryService discovery = null;

    /**
     * The Module Implementation advertisement for this service.
     */
    private Advertisement implAdvertisement = null;

    /**
```

```
 * The local Peer ID.
 */
private String localPeerID = null;

/**
 * The peer group to which the service belongs.
 */
private PeerGroup peerGroup = null;

/**
 * A unique query ID that can be used to track a query.
 */
private int queryID = 0;

/**
 * The set of listener objects registered with the service.
 */
private Vector registeredListeners = new Vector();



/**
 * PresenceServiceImpl constructor comment.
 */
public PresenceServiceImpl()
{
    super();
}

/**
 * Add a listener object to the service. When a new Presence Response
 * Message arrives, the service will notify each registered listener.
 * This method is synchronized to prevent multiple threads from
 * altering the set of registered listeners simultaneously.
 *
 * @param   listener the listener object to register with the service.
 */
```

```
    public synchronized void addListener(PresenceListener listener)
    {
        registeredListeners.addElement(listener);
    }

    /**
     * Announce presence status information to the peer group.
     *
     * @param   presenceStatus the current status to announce.
     * @param   emailAddress the user's email address.
     * @param   name the user's display name.
     */
    public void announcePresence(int presenceStatus, String emailAddress,
        String name)
    {
        if (discovery != null)
        {
            /*
            PresenceAdvertisement presenceInfo = (PresenceAdvertisement)
                AdvertisementFactory.newAdvertisement(
                    PresenceAdvertisement.getAdvertisementType());
            */
            // In some earlier versions of JXTA, registering the
            // advertisement doesn't work properly. To work around this,
            // simply instantiate the advertisement implementation
directly.
            // This is not the recommended way to get an advertisement.
            // The recommended way is shown in the commented line
            // preceeding this comment.
            PresenceAdvertisement presenceInfo = new PresenceAdv();

            // Configure the new advertisement.
            presenceInfo.setPresenceStatus(presenceStatus);
            presenceInfo.setEmailAddress(emailAddress);
            presenceInfo.setName(name);
            presenceInfo.setPeerID(localPeerID);
```

```
        try
        {
            // Publish the advertisement locally.
            discovery.publish(presenceInfo, DiscoveryService.ADV,
                DEFAULT_EXPIRATION, DEFAULT_LIFETIME);
        }
        catch (IOException e)
        {
            System.out.println("Error publishing locally: " + e);
        }

        // Publish the advertisement remotely.
        discovery.remotePublish(presenceInfo, DiscoveryService.ADV,
            DEFAULT_LIFETIME);
    }
}

/**
 * Handle notification of arriving discovery response messages,
 * determine whether the response contains presence information,
 * and, if so, dispatch the presence information to registered
 * PresenceListeners.
 *
 * @param   event the object containing the discovery response.
 */
public void discoveryEvent(DiscoveryEvent event)
{
    DiscoveryResponseMsg response = event.getResponse();

    // Extract the PresenceAdvertisement from the response.
    Enumeration responses = response.getResponses();
    while (responses.hasMoreElements())
    {
        String responseElement = (String) responses.nextElement();
```

```
            // Check for null response advertisement.
            if (null != responseElement)
            {
                // Parse the advertisement.
                try
                {
                    ByteArrayInputStream stream =
                        new ByteArrayInputStream(
                            responseElement.getBytes());

                    /*
                    PresenceAdvertisement advertisement =
                        (PresenceAdvertisement)
                            AdvertisementFactory.newAdvertisement(
                                new MimeMediaType("text/xml"), stream);
                    */

                    // In some earlier versions of JXTA, registering the
                    // advertisement doesn't work properly. To work around
                    // this, simply instantiate the advertisement
                    // implementation directly. This is not the recommended
                    // way to get an advertisement. The recommended way
                    // is shown in the commented line preceeding this
                    // comment.

                    PresenceAdvertisement advertisement =
                        new PresenceAdv(stream);
                    // Dispatch the advertisement to the registered
presence
                    // listeners.
                    Enumeration listeners =
registeredListeners.elements();
                    while (listeners.hasMoreElements())
                    {
                        PresenceListener listener =
                            (PresenceListener) listeners.nextElement();
```

```
                        // Notify the listener of the presence update.
                        listener.presenceUpdated(advertisement);
                    }
                }
                catch (IOException e)
                {
                    // Obviously not a response to our query for presence
                    // information. Ignore the error.
                    System.out.println("Error in discoveryEvent: " + e);
                }
            }
            else
            {
                System.out.println("Response advertisement is null!");
            }
        }
    }

    /**
     * Sends a query to find presence information for the user specified
     * by the given email address. Any response received by the service
     * will be dispatched to registered PresenceListener objects.
     *
     * @param   emailAddress the email address to use to find presence info.
     */
    public void findPresence(String emailAddress)
    {
        // Make sure the service has been started.
        if (discovery != null)
        {
            // Send a remote discovery for presence information.
            discovery.getRemoteAdvertisements(null,
DiscoveryService.ADV,
                tagEmailAddress, emailAddress, 0, null);
```

```
            // Do a local discovery for presence information.
            try
            {
                Enumeration enum = discovery.getLocalAdvertisements(
                    DiscoveryService.ADV, tagEmailAddress, emailAddress);

                while (enum.hasMoreElements())
                {
                    PresenceAdvertisement advertisement =
                        (PresenceAdvertisement) enum.nextElement();

                    // Dispatch the advertisement to the registered
presence
                    // listeners.
                    Enumeration listeners =
registeredListeners.elements();
                    while (listeners.hasMoreElements())
                    {
                        PresenceListener listener =
                            (PresenceListener) listeners.nextElement();

                        // Notify the listener of the presence update.
                        listener.presenceUpdated(advertisement);
                    }
                }
            }
            catch (IOException e)
            {
                System.out.println("Error in findPresence: " + e);
            }
        }
    }

    /**
     * Returns the advertisement for this service. In this case, this is
     * the ModuleImplAdvertisement passed in when the service was
```

```
     * initialized.
     *
     * @return  the advertisement describing this service.
     */
    public Advertisement getImplAdvertisement()
    {
        return implAdvertisement;
    }

    /**
     * Returns an interface used to protect this service.
     *
     * @return  the wrapper object to use to manipulate this service.
     */
    public Service getInterface()
    {
        // We don't really need to provide an interface object to protect
        // this service, so this method simply returns the service itself.
        return this;
    }

    /**
     * Initialize the service.
     *
     * @param       group the PeerGroup containing this service.
     * @param       assignedID the identifier for this service.
     * @param       implAdv the advertisement specifying this service.
     * @exception   PeerGroupException is not thrown ever by this
     *               implementation.
     */
    public void init(PeerGroup group, ID assignedID, Advertisement implAdv)
        throws PeerGroupException
    {
        // Save the module's implementation advertisement.
        implAdvertisement = (ModuleImplAdvertisement) implAdv;
```

```
    // Save a reference to the group of which that this service is
    // a part.
    peerGroup = group;

    // Get the local Peer ID.
    localPeerID = group.getPeerID().toString();

    // Register the advertisement type.
    // In some earlier versions of JXTA, registering the advertisement
    // doesn't work properly. To work around this, you can instead
    // simply instantiate the advertisement implementation directly.
    // This is not the recommended way to get an advertisement.
    // In this class, I've used the workaround in the discoveryEvent
    // and accouncePresence methods, but I've provided the
    // as well.
    /*
    AdvertisementFactory.registerAdvertisementInstance(
        PresenceAdvertisement.getAdvertisementType(),
            new PresenceAdv.Instantiator());
    */
}

/**
 * Remove a given listener object from the service. Once removed,
 * a listener will no longer be notified when a new Presence Response
 * Message arrives. This method is synchronized to prevent multiple
 * threads from altering the set of registered listeners simultaneously.
 *
 * @param   listener the listener object to unregister.
 */
public synchronized boolean removeListener(PresenceListener listener)
{
    return registeredListeners.removeElement(listener);
}

/**
```

```
     * Start the service.
     *
     * @param   args the arguments to the service. Not used.
     * @return  0 to indicate the service started.
     */
    public int startApp(String[] args)
    {
        // Now that the service is being started, set the DiscoveryService
        // object to use to publish presence information.
        discovery = peerGroup.getDiscoveryService();

        // Add ourselves as a listener.
        discovery.addDiscoveryListener(this);

        return 0;
    }
    /**
     * Stop the service.
     */
    public void stopApp()
    {
        if (discovery != null)
        {
            // Unregister ourselves as a listener.
            discovery.removeDiscoveryListener(this);
            discovery = null;

            // Empty the set of listeners.
            registeredListeners.removeAllElements();
        }
    }
}
```

Creating the Chat Service

Despite its name, the Chat service doesn't manage the chat session between two users. Instead, the Chat service is responsible for negotiating a Pipe Advertisement that can be used to establish a chat session. The Pipe Advertisement that is exchanged is used in conjunction with the `BidirectionalPipeService` to bind the input and output pipes to use from conducting the actual chat conversation.

The Initiate Chat Request Message

Before it can chat with a remote peer, a peer must request a Pipe Advertisement to establish the chat session with the remote peer. Although the Pipe Advertisement could have been included in the user's Presence Advertisement, there are a couple reasons for not doing this:

- **The functionality is unrelated—** Including the Pipe Advertisement in the Presence Advertisement would pollute the Presence Advertisement with information unrelated to conveying presence information. It would create an unnecessary link between the Presence service and the Chat service. Any developer who wanted to use the Chat service would end up having to incorporate the Presence service, even if the application didn't require Presence information.

- **A user wouldn't be able to restrict who can chat with him—** If a user's Presence Advertisement incorporated a Pipe Advertisement, anyone could start sending messages. By forcing a peer to request a Pipe Advertisement, the user's peer has the opportunity to block a chat session by not responding.

The Initiate Chat Request Message requests a Pipe Advertisement to use to establish a chat session using the XML shown in Listing 11.7.

Listing 11.7 The Initiate Chat Request Message

```
<?xml version="1.0" encoding="UTF-8"?>
<InitiateChatRequest>
  <EmailAddress> . . . </EmailAddress>
```

```
  <Name> . . . </Name>
</InitiateChatRequest>
```

The Initiate Chat Request Message contains the information that a peer receiving the request needs to determine whether to approve a chat session:

- **`EmailAddress`**— A required element containing the email address of the user making the request. The email address is used as a unique identifier for a user requesting a chat session.

- **`Name`**— An optional element containing a display name or common name for the user requesting the chat session.

When a peer receives an Initiate Chat Request Message, the peer can extract the `EmailAddress` and determine whether it wants to chat with the user requesting the chat session. If the peer wants to chat, an Initiate Chat Response Message is returned containing a Pipe Advertisement to use to establish the chat session. Otherwise, the peer does not return any response and the requesting peer cannot establish a chat session.

The Initiate Chat Request Message is broken into two classes, `InitiateChatRequestMessage` and `InitiateChatRequest`. The `InitiateChatRequestMessage` abstract class defines the majority of the functionality but leaves the implementation of the message rendering and parsing to the `InitiateChatRequest` class. The source code for `InitiateChatRequestMessage` and `InitiateChatRequest` is shown in Listings 11.8 and 11.9, respectively.

Listing 11.8 Source Code for InitiateChatRequestMessage.java

```
package com.newriders.jxta.chapter11.protocol;

import net.jxta.document.Document;
import net.jxta.document.MimeMediaType;



/**
 * An abstract class defining a request to begin a chat session. This
 * request is responsible for sending information on a peer/user requesting
```

```
 * a chat session so that the recipient can use it to determine whether
 * to send a response containing a Pipe Advertisement to use to
 * start the chat session.
 */
public abstract class InitiateChatRequestMessage
{
    /**
     * The email address of the user requesting the chat session. This
     * is used to identify the user requesting the chat session.
     */
    private String emailAddress = null;

    /**
     * A display name to use to represent the user making the request
     * during the chat session.
     */
    private String name = null;



    /**
     * Returns a Document object containing the query's document tree.
     *
     * @param       asMimeType the desired MIME type for the query
     *              rendering.
     * @return      the Document containing the query's document object
     *              tree.
     */
    public abstract Document getDocument(MimeMediaType asMimeType);

    /**
     * Retrieve the email address of the user associated with the local
     * peer.
     *
     * @return  the email address used to identify the user.
     */
    public String getEmailAddress()
```

```
    {
        return emailAddress;
    }

    /**
     *Retrieve the display name of the user associated with the local peer.
     *
     * @return  the display name used to identify the user during the
     *          chat session.
     */
    public String getName()
    {
        return name;
    }

    /**
     * Sets the email address of the user associated with the local peer.
     *
     * @param   emailAddress the email address used to identify the user.
     */
    public void setEmailAddress(String emailAddress)
    {
        this.emailAddress = emailAddress;
    }

    /**
     * Sets the display name of the user associated with the local peer.
     *
     * @param   name the display name used to identify the user during
     *          the chat session.
     */
    public void setName(String name)
    {
        this.name = name;
    }
}
```

Listing 11.9 Source Code for *InitiateChatRequest.java*

```
package com.newriders.jxta.chapter11.impl.protocol;

import java.io.InputStream;
import java.io.IOException;
import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.Element;
import net.jxta.document.Document;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;

import com.newriders.jxta.chapter11.protocol.InitiateChatRequestMessage;



/**
 * An implementation of the InitiateChatRequestMessage abstract class. This
 * class is responsible for parsing and formatting the XML document used
to
 * define a request to initiate a chat session.
 */
public class InitiateChatRequest extends InitiateChatRequestMessage
{
    /**
     * The root element for the request's XML document.
     */
    private static final String documentRootElement =
"InitiateChatRequest";

    /**
```

```
 * A convenient constant for the XML MIME type.
 */
private static final String mimeType = "text/xml";

/**
 * The element name for the email address info.
 */
private static final String tagEmailAddress = "EmailAddress";
/**
 * The element name for the display name info.
 */
private static final String tagName = "Name";



/**
 * Creates a new request object.
 */
public InitiateChatRequest()
{
    super();
}

/**
 * Creates a new Initiate Chat Request Message by parsing the
 * given stream.
 *
 * @param       stream the InputStream source of the query data.
 * @exception   IOException if the query can't be parsed from the
 *              stream.
 * @exception   IllegalArgumentException thrown if the data does not
 *              contain a Presence Query Message.
 */
public InitiateChatRequest(InputStream stream)
    throws IOException, IllegalArgumentException
{
    super();
```

```
    StructuredTextDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            new MimeMediaType(mimeType), stream);

    readDocument(document);
}

/**
 * Returns a Document object containing the request's document tree.
 *
 * @param      asMimeType the desired MIME type for the
 *             request rendering.
 * @return     the Document containing the request's document
 *             object tree.
 * @exception  IllegalArgumentException thrown if the email address
 *             is null.
 */
public Document getDocument(MimeMediaType asMimeType)
    throws IllegalArgumentException
{
    // Check that the required elements are present.
    if (null != getEmailAddress())
    {
        StructuredDocument document = (StructuredTextDocument)
            StructuredDocumentFactory.newStructuredDocument(
                asMimeType, documentRootElement);
        Element element;

        element = document.createElement(tagEmailAddress,
            getEmailAddress());
        document.appendChild(element);

        element = document.createElement(tagName, getName());
        document.appendChild(element);
```

```
            return document;
        }
        else
        {
            throw new IllegalArgumentException("Missing email address");
        }
    }

    /**
     * Parses the given document tree for the request.
     *
     * @param      document the object containing the request data.
     * @exception  IllegalArgumentException if the document is not a chat
     *             request, as expected.
     */
    public void readDocument(TextElement document)
        throws IllegalArgumentException
    {
        if (document.getName().equals(documentRootElement))
        {
            Enumeration elements = document.getChildren();

            while (elements.hasMoreElements())
            {
                TextElement element = (TextElement)
elements.nextElement();

                if (element.getName().equals(tagEmailAddress))
                {
                    setEmailAddress(element.getTextValue());
                    continue;
                }

                if (element.getName().equals(tagName))
                {
                    setName(element.getTextValue());
```

```
                    continue;
                }
            }
        }
        else
        {
            throw new IllegalArgumentException(
                "Not a InitiateChatRequest document!");
        }
    }

    /**
     * Returns an XML String representation of the request.
     *
     * @return  the XML String representing this request.
     */
    public String toString()
    {
        try
        {
            StringWriter out = new StringWriter();
            StructuredTextDocument doc =
                (StructuredTextDocument) getDocument(
                    new MimeMediaType(mimeType));
            doc.sendToWriter(out);

            return out.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

The Initiate Chat Response Message

To allow a remote peer to establish a chat session, a peer that wants to grant a chat session must generate a Pipe Advertisement and send it as part of an Initiate Chat Response Message to the requesting peer. The XML for the Initiate Chat Response Message is shown in Listing 11.10.

Listing 11.10 The Initiate Chat Response Message

```
<?xml version="1.0" encoding="UTF-8"?>
<InitiateChatRequest>
  <EmailAddress> . . . </EmailAddress>
  <Name> . . . </Name>
  <jxta:PipeAdvertisement> . . . </jxta:PipeAdvertisement>
</InitiateChatRequest>
```

The Initiate Chat Response Message provides not only the Pipe Advertisement required to establish the chat session, but also other information that the recipient peer can use:

- **`EmailAddress`**— A required element containing the email address of the user approving the request for a chat session. The email address is used as a unique identifier for the user approving the chat session.

- **`Name`**— An optional element containing a display name or common name for the user approving the chat session.

- **`jxta:PipeAdvertisement`**— A required element that contains the Pipe Advertisement to use to establish the chat session. Note that this element is actually the root of the Pipe Advertisement XML tree.

When a peer receives an Initiate Chat Response Message, it can use the Pipe Advertisement with the `BidirectionalPipeService` class to establish two-way communications and begin chatting. By using the `BidirectionalPipeService`, you avoid having to create your own protocol to handle exchanging the Pipe Advertisements required to establish two-way communications.You need to create only one Pipe

Advertisement, and the `BidirectionalPipeService` takes care of the details of exchanging Pipe Advertisements and binding input and output pipes.

The Initiate Chat Response Message is abstracted as two classes, `InitiateChatResponseMessage` and `InitiateChatResponse`. The `InitiateChatResponseMessage` abstract class defines the majority of the functionality but leaves the implementation of the message rendering and parsing to the `InitiateChatResponse` class. The source code for `InitiateChatResponseMessage` and `InitiateChatResponse` is shown in Listings 11.11 and 11.12, respectively.

Listing 11.11 Source Code for InitiateChatResponseMessage.java

```
package com.newriders.jxta.chapter11.protocol;

import net.jxta.document.Document;
import net.jxta.document.MimeMediaType;

import net.jxta.protocol.PipeAdvertisement;

/**
 * An abstract class defining a response to a request to begin a chat
 * session. This response is responsible for sending a Pipe Advertisement
 * in response to a Initiate Chat Request Message to allow a remote peer
 * to begin chatting with the local peer.
 */
public abstract class InitiateChatResponseMessage
{
    /**
     * The email address of the user associated with the local peer.
     * Used to map the user to presence information.
     */
    private String emailAddress = null;

    /**
     * A display name to use to represent the user associated with
     * the local peer during the chat session.
```

```
 */
private String name = null;

/**
 * A Pipe Advertisement to use to initiate the chat session. The
 * local peer will bind an input pipe to the pipe described by this
 * advertisement to set up the two-way chat communication channel
 * using the BidirectionalPipeService.
 */
private PipeAdvertisement pipeAdvertisement = null;



/**
 * Returns a Document object containing the response's document tree.
 *
 * @param       asMimeType the desired MIME type for the response
 *              rendering.
 * @return      the Document containing the response's document
 *              object tree.
 */
public abstract Document getDocument(MimeMediaType asMimeType);

/**
 * Retrieve the email address of the user associated with the
 * local peer.
 *
 * @return  the email address used to identify the user.
 */
public String getEmailAddress()
{
    return emailAddress;
}

/**
 * Retrieve the display name of the user associated with the local peer.
 *
```

```
     * @return  the display name used to identify the user during the
     * chat session.
     */
    public String getName()
    {
        return name;
    }

    /**
     * Returns the Pipe Advertisement object that a remote peer can use to
     * initiate the chat session.
     *
     * @return  the Pipe Advertisement to use for setting up the chat
     *          session.
     */
    public PipeAdvertisement getPipeAdvertisement()
    {
        return pipeAdvertisement;
    }

    /**
     * Sets the email address of the user associated with the local peer.
     *
     * @param   emailAddress the email address used to identify the user.
     */
    public void setEmailAddress(String emailAddress)
    {
        this.emailAddress = emailAddress;
    }

    /**
     * Sets the display name of the user associated with the local peer.
     *
     * @param   name the display name used to identify the user during the
     *          chat session.
     */
```

```
    public void setName(String name)
    {
        this.name = name;
    }

    /**
     * Sets the Pipe Advertisement object that a remote peer can use to
     * initiate the chat session.
     *
     * @param   pipeAdvertisement the Pipe Advertisement to use for setting
     *          up the chat session.
     */
    public void setPipeAdvertisement(PipeAdvertisement pipeAdvertisement)
    {
        this.pipeAdvertisement = pipeAdvertisement;
    }
}
```

Listing 11.12 Source Code for *InitiateChatResponse.java*

```
package com.newriders.jxta.chapter11.impl.protocol;

import java.io.InputStream;
import java.io.IOException;
import java.io.StringWriter;

import java.util.Enumeration;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.Element;
import net.jxta.document.Document;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentUtils;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;
```

```
import net.jxta.protocol.PipeAdvertisement;

import
com.newriders.jxta.chapter11.protocol.InitiateChatResponseMessage;


/**
 * An implementation of the InitiateChatResponseMessage abstract class.
 * This class is responsible for parsing and formatting the XML document
 * used to define a response to a request to initiate a chat session.
 */
public class InitiateChatResponse extends InitiateChatResponseMessage
{
    /**
     * The root element for the response's XML document.
     */
    private static final String documentRootElement =
        "InitiateChatResponse";

    /**
     * A convenient constant for the XML MIME type.
     */
    private static final String mimeType = "text/xml";

    /**
     * The element name for the display name info.
     */
    private static final String tagName = "Name";

    /**
     * The element name for the email address info.
     */
    private static final String tagEmailAddress = "EmailAddress";
```

```
/**
 * Creates new response object.
 */
public InitiateChatResponse()
{
    super();
}

/**
 * Creates a new Initiate Chat Response Message by parsing the given
 * stream.
 *
 * @param       stream the InputStream source of the response data.
 * @exception   IOException if the response can't be parsed from the
 *              stream.
 * @exception   IllegalArgumentException thrown if the data does not
 *              contain a Presence Response Message.
 */
public InitiateChatResponse(InputStream stream) throws IOException,
    IllegalArgumentException
{
    super();

    StructuredTextDocument document = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            new MimeMediaType(mimeType), stream);

    readDocument(document);
}
/**
 * Returns a Document object containing the response's document tree.
 *
 * @param      asMimeType the desired MIME type for the response
 *             rendering.
 * @return     the Document containing the response's document
 *             object tree.
```

```
     * @exception  IllegalArgumentException thrown if the Pipe
     *             Advertisement
     *              or the name is null.
     */
    public Document getDocument(MimeMediaType asMimeType)
        throws IllegalArgumentException
    {
        // Check that the required elements are present.
        if ((null != getPipeAdvertisement()) && (null != getName()))
        {
            StructuredDocument document = (StructuredTextDocument)
                StructuredDocumentFactory.newStructuredDocument(
                    asMimeType, documentRootElement);
            Element element;

            PipeAdvertisement pipeAdv = getPipeAdvertisement();
            if (pipeAdv != null)
            {
                StructuredTextDocument advDoc = (StructuredTextDocument)
                    pipeAdv.getDocument(asMimeType);
                StructuredDocumentUtils.copyElements(
                    document, document, advDoc);
            }

            element = document.createElement(tagName, getName());
            document.appendChild(element);

            element = document.createElement(tagEmailAddress,
                getEmailAddress());
            document.appendChild(element);

            return document;
        }
        else
        {
            throw new IllegalArgumentException("Missing pipe ID or name!");
```

```
        }
    }

    /**
     * Parses the given document tree for the response.
     *
     * @param       document the object containing the response data.
     * @exception   IllegalArgumentException if the document is not a
     *              response, as expected.
     */
    public void readDocument(TextElement document)
        throws IllegalArgumentException
    {
        if (document.getName().equals(documentRootElement))
        {
            Enumeration elements = document.getChildren();

            while (elements.hasMoreElements())
            {
                TextElement element = (TextElement)
elements.nextElement();

                if (element.getName().equals(tagName))
                {
                    setName(element.getTextValue());
                    continue;
                }

                if (element.getName().equals(tagEmailAddress))
                {
                    setEmailAddress(element.getTextValue());
                    continue;
                }

                if (element.getName().equals(
                    PipeAdvertisement.getAdvertisementType()))
```

```
                {
                    try
                    {
                        PipeAdvertisement pipeAdv = (PipeAdvertisement)
                            AdvertisementFactory.newAdvertisement(eleme
nt);
                        setPipeAdvertisement( pipeAdv );
                    }
                    catch ( ClassCastException wrongAdv )
                    {
                        throw new IllegalArgumentException(
                            "Bad pipe advertisement in advertisement" );
                    }

                    continue;
                }
            }
        }
        else
        {
            throw new IllegalArgumentException(
                "Not a InitiateChatResponse document!");
        }
    }

    /**
     * Returns an XML String representation of the response.
     *
     * @return  the XML String representing this response.
     */
    public String toString()
    {
        try
        {
            StringWriter out = new StringWriter();
            StructuredTextDocument doc = (StructuredTextDocument)
```

```
                getDocument(new MimeMediaType(mimeType));
            doc.sendToWriter(out);

            return out.toString();
        }
        catch (Exception e)
        {
            return "";
        }
    }
}
```

The Chat Message

Although the Chat service doesn't handle sending and receiving chat messages, this is probably the most appropriate place to mention the message used to send a chat message to a remote user.You could create a class to encapsulate the chat message, but in this simple implementation, only one piece of information needs to be sent to a remote user: the chat message text itself.

To send a chat message to a remote user after the pipes have been established, a peer only needs to create a new `Message` object and populate a message element named `ChatMessage` with the chat message text.

The Chat Service

The Chat service abstracts the creation of Initiate Chat Request and Response Messages and provides a simple interface that a developer can use to send these messages. As with `PresenceService`, the `ChatService` interface shown in Listing 11.13 provides a mechanism for developers to register and unregister listener objects that can be used to handle the requests and responses.

Listing 11.13 Source Code for *ChatService.java*

```
package com.newriders.jxta.chapter11.chat;
```

```
import net.jxta.protocol.PipeAdvertisement;

import net.jxta.service.Service;


/**
 * An interface for the Chat service, a service that allows peers to
 * request and approve chat sessions. This interface defines the operations
 * that a developer can expect to use to manipulate the Chat service,
 * regardless of which underlying implementation of the service is being
 * used.
 */
public interface ChatService extends Service
{
    /**
     * The module class ID for the Presence class of service.
     */
    public static final String refModuleClassID =
        "urn:jxta:uuid-F84F9397891240B496D1B5754CCC933105";


    /**
     * Add a listener object to the service. When a new Initiate Chat
     * Request or Response Message arrives, the service will notify each
     * registered listener.
     *
     * @param   listener the listener object to register with the service.
     */
    public void addListener(ChatListener listener);

    /**
     * Approve a chat session.
     *
     * @param   pipeAdvertisement the advertisement for the pipe that will
     *          be used to set up the chat session.
     * @param   emailAddress the emailAddress of the user associated with
```

```
     *          local peer.
     * @param   displayName the name of the user associated with the
     *          local peer.
     * @param   queryID the query ID to use to send to the Resolver
     *          Response Message containing the response, allowing the
     *          remote peer to match the response to an initial request.
     */
    public void approveChat(PipeAdvertisement pipeAdvertisement,
        String emailAddress, String displayName, int queryID);

    /**
     * Removes a given listener object from the service. Once removed,
     * a listener will no longer be notified when a new Initiate Chat
     * Request or Response Message arrives.
     *
     * @param   listener the listener object to unregister.
     * /
    public boolean removeListener(ChatListener listener);

    /**
     * Send a request to chat to the peer specified.
     *
     * @param   peerID the Peer ID of the remote peer to request for a chat
     *          session.
     * @param   emailAddress the email address of the user associated with
     *          the local peer.
     * @param   displayName the display name  of the user associated with
     *          the local peer.
     * @param   listener the listener to notify when a response to this
     *          request is received.
     */
    public void requestChat(String peerID, String emailAddress,
        String displayName, ChatListener listener);
}
```

For this application, we would like the main application to be capable of determining whether a request to chat should be approved. This allows the application to ignore a request from a user that isn't a part of the user's list of contacts. To accomplish this, the `ChatService` delegates the decision of whether to approve a chat request using the `ChatListener` interface shown in Listing 11.14.

Listing 11.14 Source Code for *ChatListener.java*

```
package com.newriders.jxta.chapter11.chat;

import com.newriders.jxta.chapter11.protocol.InitiateChatRequestMessage;
import
com.newriders.jxta.chapter11.protocol.InitiateChatResponseMessage;



/**
 * An interface to encapsulate an object that listens for notification
 * from the ChatService of newly arrived requests for a chat session and
 * responds to requests for a chat session.
 */
public interface ChatListener
{
    /**
     * Notify the listener that a chat session has been approved.
     *
     * @param   response the response to the request for a chat session.
     */
    public void chatApproved(InitiateChatResponseMessage response);

    /**
     * Notify the listener that a chat session has been requested.
     *
     * @param   request the object containing the chat session request info.
     * @param   queryID the query ID from the Resolver Query Message used
to
     *          send the request.
```

```
     */
    public void chatRequested(InitiateChatRequestMessage request,
        int queryID);
}
```

When the `ChatService` receives an Initiate Chat Request Message, it notifies each of the registered `ChatListener` instance's `chatRequested` methods. It is the responsibility of a listener to approve a request. When the `ChatService` receives an Initiate Chat Response Message, the registered `ChatListener` instance's `chatApproved` method handles the response, and uses its contents to begin the chat session.

For this application, the `ChatService` implementation shown in Listing 11.15 uses the Resolver service to handle the Initiate Chat Request and Response Messages.

Listing 11.15 Source Code for *ChatServiceImpl.java*

```
package com.newriders.jxta.chapter11.impl.chat;

import java.io.ByteArrayInputStream;
import java.io.IOException;

import java.util.Hashtable;
import java.util.Vector;

import net.jxta.document.Advertisement;

import net.jxta.exception.DiscardQueryException;
import net.jxta.exception.NoResponseException;
import net.jxta.exception.PeerGroupException;
import net.jxta.exception.ResendQueryException;

import net.jxta.id.ID;

import net.jxta.impl.protocol.ResolverQuery;
import net.jxta.impl.protocol.ResolverResponse;
```

```
import net.jxta.peergroup.PeerGroup;
import net.jxta.pipe.PipeID;

import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.protocol.ResolverQueryMsg;
import net.jxta.protocol.ResolverResponseMsg;

import net.jxta.resolver.QueryHandler;
import net.jxta.resolver.ResolverService;

import net.jxta.service.Service;

import com.newriders.jxta.chapter11.chat.ChatListener;
import com.newriders.jxta.chapter11.chat.ChatService;

import com.newriders.jxta.chapter11.impl.protocol.InitiateChatRequest;
import com.newriders.jxta.chapter11.impl.protocol.InitiateChatResponse;


/**
 * The implementation of the ChatService interface. This service
 * builds on top of the Resolver service to provide the functionality
 * for requesting and approving a chat session.
 */
public class ChatServiceImpl implements ChatService, QueryHandler
{
    /**
     * The Module Specification ID for the Chat service.
     */
    public static final String refModuleSpecID =
        "urn:jxta:uuid-F84F9397891240B496D1B5754CCC9331DFFD"
        + "10CDD5A140A6B8A1BC18CD65582106";

    /**
     * The set of listener objects registered with the service
```

```
     * to handle an approval to start a chat session. These
     * listeners are associated with a specific query ID used
     * to send a request to start a chat session to a remote user.
     */
    private Hashtable approvedListeners = new Hashtable();

    /**
     * The handler name used to register the Resolver handler.
     */
    private String handlerName = null;

    /**
     * The Module Implementation advertisement for this service.
     */
    private Advertisement implAdvertisement = null;

    /**
     * The local Peer ID.
     */
    private String localPeerID = null;

    /**
     * The peer group to which the service belongs.
     */
    private PeerGroup peerGroup = null;

    /**
     * A unique query ID that can be used to track a query.
     * This is constant across instances of the service on the
     * same peer to ensure queryID uniqueness for the peer.
     */
    private static int queryID = 0;

    /**
     * The set of listener objects registered with the service
     * to handle requests to start a chat session.
```

```
 */
private Vector requestListeners = new Vector();

/**
 * The Resolver service used to handle queries and responses.
 */
private ResolverService resolver = null;


/**
 * Create a new ChatServiceImpl object.
 */
public ChatServiceImpl()
{
    super();
}

/**
 * Add a listener object to the service. When a new Initiate Chat
 * Request or Response Message arrives, the service will notify each
 * registered listener. This method is synchronized to prevent multiple
 * threads from altering the set of registered listeners simultaneously.
 *
 * @param   listener the listener object to register with the service.
 */
public synchronized void addListener(ChatListener listener)
{
    requestListeners.addElement(listener);
}

/**
 * Approve a chat session.
 *
 * @param   pipeAdvertisement the advertisement for the pipe that will
 *          be used to set up the chat session.
 * @param   emailAddress the emailAddress of the user associated with
```

```
     *          local peer.
     * @param   displayName the name of the user associated with the local
     *          peer.
     * @param   queryID the query ID to use to send to the Resolver Response
     *         Message containing the response, allowing the remote peer
to
     *         match the response to an initial request.
     */
    public void approveChat(PipeAdvertisement pipeAdvertisement,
        String emailAddress, String displayName, int queryID)
    {
        // Make sure that the service has been started.
        if (resolver != null)
        {
            ResolverResponse response;

            // Create the response message and populate it with the
            // given Pipe ID.
            InitiateChatResponse reply = new InitiateChatResponse();
            reply.setPipeAdvertisement(pipeAdvertisement);
            reply.setEmailAddress(emailAddress);
            reply.setName(displayName);

            // Wrap the response message in a resolver response message.
            response = new ResolverResponse(handlerName, "JXTACRED",
                queryID, reply.toString());

            // Send the request using the Resolver service.
            resolver.sendResponse(null, response);
        }
    }

    /**
     * Returns the advertisement for this service. In this case, this is
     * the ModuleImplAdvertisement passed in when the service was
     * initialized.
```

```
     *
     * @return  the advertisement describing this service.
     */
    public Advertisement getImplAdvertisement()
    {
        return implAdvertisement;
    }

    /**
     * Returns an interface used to protect this service.
     *
     * @return  the wrapper object to use to manipulate this service.
     */
    public Service getInterface()
    {
        // We don't really need to provide an interface object to protect
        // this service, so this method simply returns the service itself.
        return this;
    }

    /**
     * Initialize the service.
     *
     * @param       group the PeerGroup containing this service.
     * @param       assignedID the identifier for this service.
     * @param       implAdv the advertisement specifying this service.
     * @exception   PeerGroupException is not thrown ever by this
     *              implementation.
     */
    public void init(PeerGroup group, ID assignedID, Advertisement implAdv)
        throws PeerGroupException
    {
        // Save a reference to the group of which that this service is
        // a part.
        peerGroup = group;
```

```
        // Use the assigned ID as the Resolver handler name.
        handlerName = assignedID.toString();

        // Save the module's implementation advertisement.
        implAdvertisement = (ModuleImplAdvertisement) implAdv;

        // Get the local Peer ID.
        localPeerID = group.getPeerID().toString();
    }

    /**
     * Process a Resolver Query Message.
     */
    public ResolverResponseMsg processQuery(ResolverQueryMsg query)
        throws IOException, NoResponseException, DiscardQueryException,
            ResendQueryException
    {
        ResolverResponse response;
        InitiateChatRequest request;

        try
        {
            // Extract the request message.
            request = new InitiateChatRequest(
                new
ByteArrayInputStream((query.getQuery()).getBytes()));
        }
        catch (Exception e)
        {
            // Not the expected format of the message.
            throw new NoResponseException();
        }

        // Notify each of the registered listeners.
        if (requestListeners.size() > 0)
        {
```

```
            ChatListener listener = null;

            for (int i = 0; i < requestListeners.size(); i++)
            {
                listener = (ChatListener) requestListeners.elementAt(i);
                listener.chatRequested(request, query.getQueryId());
            }
        }

        // Throw NoResponseException because this service will not
        // produce a InitiateChatResponse. It's the responsibility of a
        // ChatListener to decide whether to accept the request to chat
        // and inform the requestor of the Pipe ID to use to send chat
        // messages.
        throw new NoResponseException();
    }

    /**
     * Process a Resolver response message.
     *
     * @param   response a response message to be processed.
     */
    public void processResponse(ResolverResponseMsg response)
    {
        InitiateChatResponse reply;
        ChatListener listener = null;
        String responseString = response.getResponse();

        if (null != responseString)
        {
            try
            {
                // Extract the message from the Resolver response.
                reply = new InitiateChatResponse(
                    new
ByteArrayInputStream(responseString.getBytes()));
```

```
            // Notify the listener associated with the response's
            // queryID.
            listener = (ChatListener) approvedListeners.get(
                new Integer(response.getQueryId()));
            if (listener != null)
            {
                listener.chatApproved(reply);
            }
        }
        catch (Exception e)
        {
            // This is not the right type of response message, or
            // the message is improperly formed. Ignore the exception;
            // do nothing with the message.
            System.out.println("Error in response: " + e);
        }
    }
}

/**
 * Remove a given listener object from the service. Once removed,
 * a listener will no longer be notified when a new Initiate Chat
 * Request or Response Message arrives. This method is synchronized to
 * prevent multiple threads from altering the set of registered
 * listeners simultaneously.
 *
 * @param   listener the listener object to unregister.
 */
public synchronized boolean removeListener(ChatListener listener)
{
    return requestListeners.removeElement(listener);
}

/**
 * Send a request to chat to the peer specified.
```

```
     *
     * @param   peerID the Peer ID of the remote peer to request for a
     *          chat session.
     * @param   emailAddress the email address of the user associated
     *          with the local peer.
     * @param   displayName the display name of the user associated with
     *          the local peer.
     * @param   listener the listener to notify when a response to this
     *          request is received.
     */
    public void requestChat(String peerID, String emailAddress,
        String displayName, ChatListener listener)
    {
        // Make sure that the service has been started.
        if (resolver != null)
        {
            // Create the request object.
            String localPeerID = peerGroup.getPeerID().toString();
            InitiateChatRequest request = new InitiateChatRequest();

            // Configure the request.
            request.setEmailAddress(emailAddress);
            request.setName(displayName);

            // Wrap the query in a Resolver Query Message.
            ResolverQuery query = new ResolverQuery(handlerName,
                "JXTACRED", localPeerID, request.toString(), queryID++);

            // Add the given listener to the set of approved listeners.
            // This will be used to ensure that only responses to actual
            // queries sent by this service will be passed to the given
            // listener.
            approvedListeners.put(
                new Integer(query.getQueryId()), listener);

            // Send the request to the peer using the Resolver service.
```

```
        resolver.sendQuery(peerID, query);
    }
}

/**
 * Start the service.
 *
 * @param   args the arguments to the service. Not used.
 * @return  0 to indicate the service started.
 */
public int startApp(String[] args)
{
    // Now that the service is being started, set the ResolverService
    // object to use handle our queries and send responses.
    resolver = peerGroup.getResolverService();

    // Add ourselves as a handler using the uniquely constructed
    // handler name.
    resolver.registerHandler(handlerName, this);

    return 0;
}

/**
 * Stop the service.
 */
public void stopApp()
{
    if (resolver != null)
    {
        // Unregister ourselves as a listener.
        resolver.unregisterHandler(handlerName);
        resolver = null;

        // Empty the set of request and approved listeners.
        requestListeners.removeAllElements();
```

```
            }
        }
}
```

The JXTA Messenger Application

The JXTA Messenger application is the chat application that the end user will see and use to conduct a chat session with a remote user. The application itself consists of two pieces: an application module to show the user interface and a main application to handle configuring and creating the peer group.

The User Interface

The `ExampleService` example developed in Chapter 10, "Peer Groups and Services," had no user interface of its own to allow a user to interact with the service. The `ExampleServiceTest` class provided the user interface and interacted with the peer group's `ExampleService` instance to provide functionality. Although you could do the same thing in the JXTA Messenger, it would be better to wrap up the entire user interface as an application that starts when the peer group starts.

Fortunately, the reference implementation of JXTA provides the capability to add an application to a peer group's Module Implementation Advertisement parameters. When the `ExampleService` implementation was added to the service parameters for the peer group created in `ExampleServiceTest`, it involved code similar to the code shown in Listing 11.16.

Listing 11.16 Adding a Service to the Peer Group Parameters

```
ModuleImplAdvertisement implAdv =
    netPeerGroup.getAllPurposePeerGroupImplAdvertisement();
StdPeerGroupParamAdv params = new
StdPeerGroupParamAdv(implAdv.getParam());
Hashtable services = params.getServices();
services.put(moduleClassAdv.getModuleClassID(), moduleImplAdv);
params.setServices(services);
implAdv.setParam((StructuredDocument) params.getDocument(
```

```
    new MimeMediaType("text", "xml")));
```

This code added the service specified by the `moduleImplAdv` Module Implementation Advertisement to the set of parameters in `implAdv`, the peer group's Module Implementation Advertisement. The new service is added to the parameters' `services Hashtable` using the service's Module Class ID as a key.

In the reference implementation, any class that implements the `net.jxta.platform.Application` interface can be added to the peer group's Module Implementation Advertisement parameters in a similar fashion. Instead of adding to the parameters' set of services, add the application to the parameters' set of applications.

```
Hashtable applications = params.getApps ();
. . .
params.setApps(applications);
```

Only two real differences exist between an application module and a service module in the reference implementation:

- **Which interface the module implements—** An application module implements the `net.jxta.platform.Application` interface, whereas a service module implements the `net.jxta.service.Service` interface. Both inter-faces extend the `net.jxta.platform.Module` interface, and only `Service` adds methods not found in `Module`.

- **When the module's `startApp` method is called—** Each of a peer group's services has its `startApp` method called when the peer group, which is also a module, is initialized using the `PeerGroup.init` method. Each of the peer group's applications has its `startApp` method called when the peer group is started using the `PeerGroup.startApp` method.

So, rather than try to have the main application figure out when the peer group has started, you should place the user interface in a separate class that implements the `Application` interface. That way, the user interface automatically is notified via its `startApp` method when the peer group containing the Presence and Chat services has been started.

The `BuddyList` class, shown in Listing 11.17, handles the main user interface for the JXTA Messenger. The `BuddyList` provides a user interface that displays a list of buddies that the user is monitoring for presence information.

Listing 11.17 Source Code for *BuddyList.java*

```
package com.newriders.jxta.chapter11;

import java.util.Enumeration;
import java.io.IOException;

import java.util.Hashtable;

import java.awt.Container;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import java.net.MalformedURLException;
import java.net.URL;
import java.net.UnknownServiceException;

import javax.swing.BorderFactory;
import javax.swing.ButtonGroup;
import javax.swing.DefaultListModel;
import javax.swing.JCheckBoxMenuItem;
import javax.swing.JFrame;
import javax.swing.JList;
import javax.swing.JMenuBar;
import javax.swing.JMenu;
import javax.swing.JMenuItem;
import javax.swing.JOptionPane;
import javax.swing.JScrollPane;
```

```
import javax.swing.border.Border;

import javax.swing.event.ListSelectionEvent;
import javax.swing.event.ListSelectionListener;

import net.jxta.document.Advertisement;

import net.jxta.exception.PeerGroupException;
import net.jxta.exception.ServiceNotFoundException;

import net.jxta.id.ID;
import net.jxta.id.IDFactory;

import net.jxta.impl.util.BidirectionalPipeService;

import net.jxta.peergroup.PeerGroup;

import net.jxta.platform.Application;
import net.jxta.platform.ModuleClassID;

import net.jxta.protocol.PipeAdvertisement;

import com.newriders.jxta.chapter11.chat.ChatListener;
import com.newriders.jxta.chapter11.chat.ChatService;

import com.newriders.jxta.chapter11.impl.protocol.InitiateChatResponse;

import com.newriders.jxta.chapter11.presence.PresenceListener;
import com.newriders.jxta.chapter11.presence.PresenceService;

import com.newriders.jxta.chapter11.protocol.InitiateChatRequestMessage;
import
com.newriders.jxta.chapter11.protocol.InitiateChatResponseMessage;
import com.newriders.jxta.chapter11.protocol.PresenceAdvertisement;
/**
 * A user interface application to show a buddy list and their current
```

```
 * status using the Presence service. The application also allows the user
 * to initiate a chat session using the Chat service.
 */
public class BuddyList extends JFrame implements Application,
    ActionListener, PresenceListener
{
    /**
     * The Module Class ID for the BuddyList application.
     */
    public static final String refModuleClassID =
        "urn:jxta:uuid-E340D55F97E141C9B46FB2B108D8C2B705";

    /**
     * The Module Specification ID for the BuddyList application.
     */
    public static final String refModuleSpecID =
        "urn:jxta:uuid-E340D55F97E141C9B46FB2B108D8C2B7"
        + "581B0312E66046B6BB96BF6A2EC5F27906";

    /**
     * A list to display the set of "approved" buddies and their
     * presence status.
     */
    private JList buddies = new JList();

    /**
     * The data model for the list widget.
     */
    private DefaultListModel buddiesData = new DefaultListModel();

    /**
     * The peer group to which the service belongs.
     */
    private PeerGroup peerGroup = null;

    /**
```

```
 * The Presence service used to update other users of this user's
 * current presence status.
 */
private PresenceService presence = null;
/**
 * The Chat service used to handle requesting chat sessions and
 * respond with approvals.
 */
private ChatService chat = null;

/**
 * A set of buddy email addresses, indexed by display name.
 */
private Hashtable buddyNames = new Hashtable();

/**
 * A set of buddy display names, indexed by email address.
 */
private Hashtable buddyEmailAddresses = new Hashtable();

/**
 * A set of buddy Peer IDs, indexed by email address.
 */
private Hashtable buddyPeerIDs = new Hashtable();

/**
 * The local user's email address.
 */
private String emailAddress = null;

/**
 * The local user's display name.
 */
private String displayName = null;

/**
```

```
 * The local user's current presence status.
 */
private int presenceStatus = PresenceService.OFFLINE;

/**
 * A handler to use for handling chat requests and approvals.
 */
private ChatHandler chatHandler = new ChatHandler();
/**
 * A simple menu handler to deal with the "Set Name..." menu item.
 */
public class SetNameHandler implements ActionListener
{
    /**
     * Handles the "Set Name..." menu item.
     *
     * @param   e the event for the menu item.
     */
    public void actionPerformed(ActionEvent e)
    {
        String newDisplayName = JOptionPane.showInputDialog(null,
            "Enter a new display name :",
            "Set Display Name...", JOptionPane.QUESTION_MESSAGE);

        if ((null != newDisplayName) && (0 < newDisplayName.length()))
        {
            // Announce change in presence status.
            presence.announcePresence(
                presenceStatus, emailAddress, newDisplayName);

            displayName = newDisplayName;
        }
    }
}

/**
```

```
     * A simple menu handler to deal with the "Add Buddy..." menu item.
     */
    public class AddBuddyHandler implements ActionListener
    {
        /**
         * Handles the "Add Buddy..." menu item.
         *
         * @param   e the event for the menu item.
         */
        public void actionPerformed(ActionEvent e)
        {
            String buddy = JOptionPane.showInputDialog(null,
                "Enter the email address of your buddy:",
                "Add Buddy...", JOptionPane.QUESTION_MESSAGE);
                if ((null != buddy) && (0 < buddy.length()))
                {
                    // Ensure that the buddy isn't already in our list.
                    if (null == buddyEmailAddresses.get(buddy))
                    {
                        // We should really validate the email address, but
                        // for simplicity we'll just add it to the list
                        // marking the buddy as "offline". Use the email
                        // address as the buddy display name until we
discover
                        //  presence information.
                        add(buddy, buddy, PresenceService.OFFLINE);

                        // Find the presence information for the buddy.
                        presence.findPresence(buddy);
                }
                else
                {
                    JOptionPane.showMessageDialog(null,
                        "A buddy with that email address already exists!",
                        "Buddy Exists!", JOptionPane.ERROR_MESSAGE);
                }
```

```
            }
        }
    }

    /**
     * A handler to deal with the application dialog being closed or the
     * "Quit" menu item being activated.
     */
    public class QuitHandler extends WindowAdapter implements
ActionListener
    {
        /**
         * Handles the "Quit" menu item.
         *
         * @param   e the event for the menu item.
         */
        public void actionPerformed(ActionEvent e)
        {
            quit();
        }
        /**
         * Handles the window's system Close button.
         *
         * @param   e the event for the window closing.
         */
        public void windowClosing(WindowEvent e)
        {
            quit();
        }

        /**
         * Quits the application.
         */
        private void quit()
        {
            // Announce change in presence status.
```

```
            presenceStatus = PresenceService.OFFLINE;
            presence.announcePresence(
                presenceStatus, emailAddress, displayName);

            System.exit(0);
        }
    }

    /**
     * Handles the local user updating presence information.
     */
    public class StatusChangeHandler implements ActionListener
    {
        /**
          * Handles the "My Status" submenu items.
          *
          * @param   e the event for the menu item.
          */
        public void actionPerformed(ActionEvent e)
        {
            int newPresenceStatus = PresenceService.OFFLINE;
            String presenceString =
                ((JCheckBoxMenuItem) e.getSource()).getText();

            if (presenceString.equals("Offline"))
            {
                newPresenceStatus = PresenceService.OFFLINE;
            }
            else if (presenceString.equals("Online"))
            {
                newPresenceStatus = PresenceService.ONLINE;
            }
            else if (presenceString.equals("Busy"))
            {
                newPresenceStatus = PresenceService.BUSY;
            }
```

```
        else if (presenceString.equals("Away"))
        {
            newPresenceStatus = PresenceService.AWAY;
        }

        if (newPresenceStatus != presenceStatus)
        {
            // Announce change in presence status.
            presence.announcePresence(
                newPresenceStatus, emailAddress, displayName);
            presenceStatus = newPresenceStatus;
        }
    }
}

/**
 * A simple handler to deal with spawning a chat window when a chat
 * request is approved or for handling incoming chat requests.
 */
public class ChatHandler implements ChatListener
{
/**
 * Handles an approval for a previously generated chat request.
 * Displays the ChatDialog and handles establishing the two-
 * way communication channel.
 *
 * @param   response the response object containing the Pipe
 *          Advertisement to use to establish two-way communication.
 */
public void chatApproved(InitiateChatResponseMessage response)
    {
        ChatDialog chatDialog = null;

        // Extract the Pipe Advertisement from the chat response.
        PipeAdvertisement pipeAdv = response.getPipeAdvertisement();
```

```
if (null != pipeAdv)
{
    // Create a bidirectional pipe.
    BidirectionalPipeService pipeService =
        new BidirectionalPipeService(peerGroup);
    BidirectionalPipeService.Pipe pipe = null;
    String buddyName = null;

    while (null == pipe)
    {
        // We just loop here.
        try
        {
            pipe = pipeService.connect(pipeAdv, 120000);
        }
        catch (IOException e)
        {
            // Do nothing.
            System.out.println("Connect error:" + e);
        }
    }

    // Get the buddy's display name.
    buddyName = response.getName();
    if (buddyName == null)
    {
        buddyName = response.getEmailAddress();
    }

    // Create the conversation GUI, and show it.
    chatDialog = new ChatDialog(buddyName, displayName,
        peerGroup.getPipeService(), pipe.getInputPipe(),
        pipe.getOutputPipe());
    chatDialog.show();
}
else
```

```
    {
        JOptionPane.showMessageDialog(null,
            "Buddy's reply is missing pipe advertisement!",
            "Unable To Chat!", JOptionPane.ERROR_MESSAGE);
    }
}

/**
 * Handles an incoming request for a chat session. Checks that
 * the incoming request comes from a known buddy and responds
 * with an approval message. Also prepares two-way communications
 * and the chat user interface.
 *
 * @param   request the request for a chat session.
 * @param   queryID the query ID to be used to send a response
 *          using the Resolver service.
 */
public void chatRequested(InitiateChatRequestMessage request,
    int queryID)
{
    String buddyEmailAddress = request.getEmailAddress();

    // Check who is making the request against the list of
    // approved chat buddies.
    if (null != buddyEmailAddresses.get(buddyEmailAddress))
    {
        ChatDialog chatDialog = null;
        String buddyName = null;

        // If the request is part of the approved buddies, then
        // approve the chat request.
        BidirectionalPipeService pipeService =
            new BidirectionalPipeService(peerGroup);
        BidirectionalPipeService.Pipe pipe = null;

        // Create an accept pipe to use to create an input pipe and
```

```
// listen for connections.
try
{
    BidirectionalPipeService.AcceptPipe acceptPipe =
        pipeService.bind("JXTA Messenger Pipe");
// Extract the Pipe Advertisement and the Pipe ID.
PipeAdvertisement pipeAdv =
    acceptPipe.getAdvertisement();

// Send the approval response.
chat.approveChat(pipeAdv, emailAddress,
    displayName, queryID);

// "Accept" a connection, meaning set up the input pipe
// and listen for messages. Set this object as the
// MessageListener so that we can handle incoming
// messages without having to spawn a thread to call
// waitForMessage on the input pipe.
while (null == pipe)
{
    // We just loop here.
    try
    {
        pipe = acceptPipe.accept(1200000);
    }
    catch (InterruptedException e)
    {
        // Do nothing.
        System.out.println("Interrupted: " + e);
    }
}

// Get the buddy's display name.
buddyName = request.getName();
if (buddyName == null)
{
```

```
                buddyName = request.getEmailAddress();
            }

            // Create the conversation GUI.
            chatDialog = new ChatDialog(buddyName, displayName,
                peerGroup.getPipeService(), pipe.getInputPipe(),
                pipe.getOutputPipe());
        }
        catch (IOException e2)
        {
                System.out.println("Error in chatRequested: " +  e2);
            }
        }
    }
}

/**
 * Creates a new BuddyList object.
 */
public BuddyList()
{
    super("JXTA Messenger");
}

/**
 * Handles the user interface event used to trigger
 * a chat session with a buddy from the list.
 *
 * @param   e the event being handled.
 */
public void actionPerformed(ActionEvent e)
{
    if (null != chat)
    {
        if (-1 != buddies.getSelectedIndex())
        {
```

```
                String buddyEmailAddress = null;
                String peerID = null;
                String buddyName =
                    (String)
buddiesData.get(buddies.getSelectedIndex());

                // Extract the actual buddy name (without the
                // presence string).
                buddyName = buddyName.substring(0, buddyName.indexOf("
("));

                // Figure out the email address and Peer ID.
                buddyEmailAddress = (String) buddyNames.get(buddyName);
                peerID = (String) buddyPeerIDs.get(buddyEmailAddress);

                if ((null != buddyEmailAddress) && (null != peerID))
                {
                    // It would be best to time out due to lack of approval
                    // for chat session, but for simplicity we'll just
                    // hope that the request gets through. In a full-fledged
                    // application, it would be better to time out and
retry.
                    chat.requestChat(peerID, emailAddress, displayName,
                        chatHandler);
                }
                else
                {
                    JOptionPane.showMessageDialog(null,
                        "Could not find buddy!", "Unable To Find Buddy",
                        JOptionPane.ERROR_MESSAGE);
                }
            }
            else
            {
                JOptionPane.showMessageDialog(null,
                    "No buddy selected!", "Select A Buddy First!",
```

```
                JOptionPane.ERROR_MESSAGE);
        }
    }
}

/**
 * Add a buddy to the list of buddies.
 *
 * @param   buddyDisplayName the display name for the buddy.
 * @param   buddyEmailAddress the email address for the buddy.
 * @param   status the initial presence status for the buddy.
 */
public void add(String buddyDisplayName, String buddyEmailAddress,
    int status)
{
    String buddyStatus;

    buddyNames.put(buddyDisplayName, buddyEmailAddress);
    buddyEmailAddresses.put(buddyEmailAddress, buddyDisplayName);

    // Construct the list entry from both the displayName and
    // the current status.
    buddyStatus = buddyDisplayName + " ";
    switch (status)
    {
        case PresenceService.OFFLINE:
        {
            buddyStatus += "(Offline)";
            break;
        }

        case PresenceService.ONLINE:
        {
            buddyStatus += "(Online)";
            break;
        }
```

```
        case PresenceService.BUSY:
        {
            buddyStatus += "(Busy)";
            break;
        }

        case PresenceService.AWAY:
        {
            buddyStatus += "(Away)";
            break;
        }

        default:
        {
            buddyStatus += "(Offline)";
            break;
        }
    }
    buddiesData.addElement(buddyStatus);
}

/**
 * Initialize the BuddyList application for the peer group.
 *
 * @param       group the peer group containing this application.
 * @param       ID the assigned ID for the application.
 * @param       implAdv the Module Implementation Advertisement for
 *              the BuddyList application.
 * @exception   PeerGroupException never thrown in this implementation.
 */
public void init(PeerGroup group, ID assignedID, Advertisement implAdv)
    throws PeerGroupException
{
    this.peerGroup = group;
```

```
    // Initialize the user interface for the buddy list.
    initializeUserInterface();
}

/**
 * Initialize the menu for the BuddyList user interface.
 */
public JMenuBar initializeMenu()
{
    JMenuBar menuBar = new JMenuBar();
    JMenu actionsMenu = new JMenu("Actions");
    JMenu myStatusMenuItem = new JMenu("My Status");
    JCheckBoxMenuItem offline = new JCheckBoxMenuItem("Offline");
    JCheckBoxMenuItem online = new JCheckBoxMenuItem("Online", true);
    JCheckBoxMenuItem busy = new JCheckBoxMenuItem("Busy");
    JCheckBoxMenuItem away = new JCheckBoxMenuItem("Away");
    JMenuItem addBuddyMenuItem = new JMenuItem("Add Buddy...");
    JMenuItem chatBuddyMenuItem = new JMenuItem("Chat With Buddy...");
    JMenuItem setNameMenuItem = new JMenuItem("Set Name...");
    JMenuItem quitMenuItem = new JMenuItem("Quit");
    ButtonGroup group = new ButtonGroup();

    // Configure the status menu.
    myStatusMenuItem.add(offline);
    group.add(offline);
    myStatusMenuItem.add(online);
    group.add(online);
    myStatusMenuItem.add(busy);
    group.add(busy);
    myStatusMenuItem.add(away);
    group.add(away);

    // Configure the listeners for the menu items.
    addBuddyMenuItem.addActionListener(new AddBuddyHandler());
    chatBuddyMenuItem.addActionListener(this);
    setNameMenuItem.addActionListener(new SetNameHandler());
```

```
    quitMenuItem.addActionListener(new QuitHandler());
    offline.addActionListener(new StatusChangeHandler());
    online.addActionListener(new StatusChangeHandler());
    busy.addActionListener(new StatusChangeHandler());
    away.addActionListener(new StatusChangeHandler());

    // Add the menu items to the menus.
    actionsMenu.add(myStatusMenuItem);
    actionsMenu.add(addBuddyMenuItem);
    actionsMenu.add(chatBuddyMenuItem);
    actionsMenu.add(setNameMenuItem);
    actionsMenu.addSeparator();
    actionsMenu.add(quitMenuItem);

    // Add the menus to the menu bar.
    menuBar.add(actionsMenu);

    return menuBar;
}

/**
 * Initialize the main user interface for the BuddyList application.
 */
public void initializeUserInterface()
{
    Container framePanel = getContentPane();

    // Set a border for the list of buddies.
    Border border = BorderFactory.createTitledBorder("Buddies");

    // Configure menu.
    setJMenuBar(initializeMenu());

    // Add the initialized inner panel to the frame panel.
    framePanel.add(buddies);
```

```
    // Pack the frame, preparing it for display.
    pack();
    setSize(200, 300);

    // Add a listener to handle quitting the application when
    // the window is closed using the system menu.
    addWindowListener(new QuitHandler());

    // Add ourselves as a listener to the list, and set
    // the model to supply the data for the list.
    buddies.setModel(buddiesData);
}

/**
 * Handles updating the user interface when new presence
 * information arrives.
 *
 * @param   presenceInfo the Presence Advertisement containing the
 *          newly arrived presence information.
 */
public void presenceUpdated(PresenceAdvertisement presenceInfo)
{
    String buddyName = null;

    // First check that this buddy is someone we're interested
    // in displaying presence information for.
    buddyName =
        (String) buddyEmailAddresses.get(
            presenceInfo.getEmailAddress());
    if (null != buddyName)
    {
        // Add the Peer ID to our Hashtable of Peer IDs.
        buddyPeerIDs.put(presenceInfo.getEmailAddress(),
            presenceInfo.getPeerID());

        // Update the list entry.
```

```
for (int i = 0; i < buddiesData.getSize(); i++)
{
    String currentBuddy = (String) buddiesData.get(i);

    // See if this is the right buddy to update.
    if (currentBuddy.indexOf(buddyName) != -1)
    {
        // Update the buddy name (in case it changed).
        String buddyDisplayName = presenceInfo.getName();
        String buddyEmailAddress =
            presenceInfo.getEmailAddress();
        if (null == buddyDisplayName)
        {
            buddyDisplayName = buddyEmailAddress;
        }
        // Fix hashtables to reflect changes in name.
        buddyNames.remove(buddyName);
        buddyNames.put(buddyDisplayName, buddyEmailAddress);
        buddyEmailAddresses.remove(buddyEmailAddress);
        buddyEmailAddresses.put(buddyEmailAddress,
            buddyDisplayName);

        // Update the list element to reflect the
        // presence change.
        buddyDisplayName += " ";
        switch (presenceInfo.getPresenceStatus())
        {
            case PresenceService.OFFLINE:
            {
                buddyDisplayName += "(Offline)";
                break;
            }

            case PresenceService.ONLINE:
            {
                buddyDisplayName += "(Online)";
```

```
                        break;
                    }

                    case PresenceService.BUSY:
                    {
                        buddyDisplayName += "(Busy)";
                        break;
                    }

                    case PresenceService.AWAY:
                    {
                        buddyDisplayName += "(Away)";
                        break;
                    }

                    default:
                    {
                        buddyDisplayName += "(Offline)";
                        break;
                    }
                }
                buddiesData.set(i, buddyDisplayName);
            }
        }
    }
}

/**
 * Starts the BuddyList application.
 *
 * @param   args the arguments to use to start the application.
 * @return  a status value.
 */
public int startApp(String[] args)
{
    int result = 0;
```

```
        try
        {
            // Find the Presence service on the peer group.
            ModuleClassID classID = (ModuleClassID) IDFactory.fromURL(
                new URL((PresenceService.refModuleClassID)));
            presence = (PresenceService)
peerGroup.lookupService(classID);

            // Register ourselves as a presence listener.
            presence.addListener(this);

            // Find the Chat service on the peer group.
            classID = (ModuleClassID) IDFactory.fromURL(
                new URL((ChatService.refModuleClassID)));
            chat = (ChatService) peerGroup.lookupService(classID);

            // Register ourselves as a chat request listener.
            chat.addListener(chatHandler);

            // Prompt the user to enter his own user information.
            do
            {
                emailAddress = JOptionPane.showInputDialog(null,
                    "What is your email address?",
                    "Configuration: Step 1 of 2",
                    JOptionPane.QUESTION_MESSAGE);
            } while (null == emailAddress);
            do
            {
                displayName = JOptionPane.showInputDialog(null,
                    "Enter a display name:",
                    "Configuration: Step 2 of 2",
                    JOptionPane.QUESTION_MESSAGE);
            } while (null == displayName);
```

```
        // Announce that we are online.
        presenceStatus = PresenceService.ONLINE;
        presence.announcePresence(presenceStatus, emailAddress,
            displayName);

        // Show the user interface.
        initializeUserInterface();
        setVisible(true);
    }
    catch (Exception e)
    {
        result = 1;
    }

    return result;
}

/**
 * Stop the application.
 */
public void stopApp()
{
    if (null != presence)
    {
        // Remove ourselves as a presence listener.
        presence.removeListener(this);

        presence = null;
    }

    if (null != chat)
    {
        // Remove ourselves as a chat request listener.
        chat.removeListener(chatHandler);

        chat = null;
```

```
        }

        // Hide the user interface.
        setVisible(false);
    }
}
```

The `BuddyList` class implements the `PresenceListener` interface, allowing it to update the user interface as updated presence information is received. The `BuddyList` class also implements the `ChatListener` interface so that it can ensure that requests to start a chat session are approved only for users who are in the user's list of buddies.

The `BuddyList` class relies on a separate class to handle presenting a user interface for a chat session. The `ChatDialog` class, shown in Listing 11.18, is displayed by `BuddyList` when a chat session has been successfully established.

Listing 11.18 Source Code for *ChatDialog.java*

```
package com.newriders.jxta.chapter11;

import java.awt.BorderLayout;
import java.awt.Container;
import java.awt.GridBagConstraints;
import java.awt.GridBagLayout;

import java.io.IOException;

import java.awt.event.ActionEvent;
import java.awt.event.ActionListener;
import java.awt.event.KeyEvent;
import java.awt.event.KeyListener;
import java.awt.event.WindowAdapter;
import java.awt.event.WindowEvent;

import javax.swing.BorderFactory;
import javax.swing.JButton;
```

```
import javax.swing.JFrame;
import javax.swing.JPanel;
import javax.swing.JScrollPane;
import javax.swing.JTextArea;

import net.jxta.endpoint.Message;

import net.jxta.pipe.InputPipe;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.PipeService;



/**
 * A class to display the chat session user interface and handle the pipes
 * used to send and receive chat messages.
 */
public class ChatDialog extends JFrame implements ActionListener,
    KeyListener
{
    /**
     * The text area used to enter a chat message to send to a remote user.
     */
    private JTextArea message = new JTextArea(3, 20);

    /**
     * The text area to show the incoming and outgoing chat messages in
     * the conversation.
     */
    private JTextArea conversation = new JTextArea(12, 20);

    /**
     * The input pipe being used to receive chat messages.
     */
    private InputPipe inputPipe = null;

    /**
```

```
 * The output pipe being used to send chat messages.
 */
private OutputPipe outputPipe = null;

/**
* The name of the remote buddy in the conversation.
*/
private String buddyName = null;

/**
 * The name of the local user in the conversation.
 */
private String displayName = null;

/**
 * The pipe service to use to create Message objects.
 */
private PipeService pipe = null;

/**
 * A thread to handle receiving messages and updating the user
 * interface.
 */
private MessageReader reader = null;

/**
 * A handler class to deal with closing the window.
 */
public class WindowHandler extends WindowAdapter
{
    /**
     * Handles the window closing.
     *
     * @param   e the object with details of the window event.
     */
    public void windowClosing(WindowEvent e)
```

```
    {
        if (reader != null)
        {
            reader.stop();
        }

        setVisible(false);
    }
}

/**
 * A simple thread to handle reading messages from the input pipe and
 * updating the user interface.
 */
public class MessageReader extends Thread
{
    /**
     * The main thread loop.
     */
    public void run()
    {
        while (true)
        {
            try
            {
                Message messageObj = inputPipe.waitForMessage();

                // Make sure that the dialog is visible.
                setVisible(true);

                // Extract the Chat Message.
                StringBuffer chatMessage =
                    new StringBuffer(
                        messageObj.getString("ChatMessage"));

                // Update the user interface.
```

```
                StringBuffer conversationText =
                    new StringBuffer(conversation.getText());
                conversationText.append("\n");
                conversationText.append(buddyName).append("> ");
                conversationText.append(chatMessage);
                conversation.setText(conversationText.toString());
            }
            catch (Exception e)
            {
                System.out.println("Error...: " + e);
            }
        }
    }
}

/**
 * Create a new window to handle a conversation with a remote user.
 *
 * @param   buddyName the display name for the remote user in the
 *          chat session.
 * @param   displayName the display name for the local user in the
 *          chat session.
 * @param   pipe the pipe service to use to create messages.
 * @param   inputPipe the pipe to use to receive messages.
 * @param   outputPipe the pipe to use to send messages.
 */
public ChatDialog(String buddyName, String displayName,
    PipeService pipe, InputPipe inputPipe, OutputPipe outputPipe)
{
    super();

    this.pipe = pipe;
    this.inputPipe = inputPipe;
    this.outputPipe = outputPipe;
    this.buddyName = buddyName;
    this.displayName = displayName;
```

```
    // Initialize the user interface.
    initializeUserInterface();

    // Set the title of the dialog.
    setTitle("Conversation - " + buddyName);

    reader = new MessageReader();
    reader.start();
}

/**
 * Handles the "Send" button.
 *
 * @param   e the event corresponding to the button being pressed.
 */
public void actionPerformed(ActionEvent e)
{
    sendMessage();
}

/**
 * Initializes the dialog's user interface.
 */
public void initializeUserInterface()
{
    Container framePanel = getContentPane();
    JPanel conversationPanel = new JPanel();
    JPanel sendPanel = new JPanel();
    JButton sendButton = new JButton("Send!");
    GridBagLayout layout = new GridBagLayout();
    JScrollPane    messagePane = new JScrollPane(message);

    GridBagConstraints constraints = new GridBagConstraints();

    constraints.gridx = 0;
```

```
constraints.gridy = 0;
constraints.gridwidth = 1;
constraints.gridheight = 1;
constraints.anchor = GridBagConstraints.WEST;
constraints.weightx = 1;
constraints.weighty = 1;
constraints.fill = GridBagConstraints.HORIZONTAL;
layout.addLayoutComponent(messagePane, constraints);

constraints.gridx = 1;
constraints.gridy = 0;
constraints.gridwidth = 1;
constraints.gridheight = 1;
constraints.anchor = GridBagConstraints.WEST;
constraints.weightx = 0.1;
constraints.weighty = 1;
constraints.fill = GridBagConstraints.BOTH;
layout.addLayoutComponent(sendButton, constraints);

sendPanel.setLayout(layout);
sendPanel.setBorder(BorderFactory.createTitledBorder(
    "Compose A Message:"));
sendPanel.add(messagePane);
sendPanel.add(sendButton);

conversationPanel.setLayout(new BorderLayout());
conversationPanel.setBorder(
    BorderFactory.createTitledBorder("Conversation:"));
conversationPanel.add(new JScrollPane(conversation),
    BorderLayout.CENTER);
conversation.setEditable(false);

constraints.gridx = 0;
constraints.gridy = 0;
constraints.gridwidth = 1;
constraints.gridheight = 1;
```

```
    constraints.anchor = GridBagConstraints.NORTH;
    constraints.weightx = 1;
    constraints.weighty = 1;
    constraints.fill = GridBagConstraints.BOTH;
    layout.addLayoutComponent(conversationPanel, constraints);

    constraints.gridx = 0;
    constraints.gridy = 1;
    constraints.gridwidth = 1;
    constraints.gridheight = 1;
    constraints.anchor = GridBagConstraints.NORTH;
    constraints.weightx = 1;
    constraints.weighty = 0;
    constraints.fill = GridBagConstraints.BOTH;
    layout.addLayoutComponent(sendPanel, constraints);

    framePanel.setLayout(layout);
    framePanel.add(conversationPanel);
    framePanel.add(sendPanel);

    sendButton.addActionListener(this);
    message.addKeyListener(this);

    pack();
}

/**
 * Invoked when a key has been pressed.
 *
 * @param   e the event describing the key event.
 */
public void keyPressed(KeyEvent e)
{
    // Do nothing. Only need keyReleased method from KeyListener.
}
```

```
/**
 * Invoked when a key has been released.
 *
 * @param   e the event describing the key event.
 */
public void keyReleased(KeyEvent e)
{
    // Handle the user pressing Return in the message composition
    // text area.
    if (KeyEvent.VK_ENTER == e.getKeyCode())
    {
        sendMessage();
    }
}

/**
 * Invoked when a key has been typed.
 *
 * @param   e the event describing the key event.
 */
public void keyTyped(KeyEvent e)
{
    // Do nothing. Only need keyReleased method from KeyListener.
}

/**
 * Send the message in the message composition text area to the
 * remote user.
 */
public void sendMessage()
{
    StringBuffer conversationText =
        new StringBuffer(conversation.getText());
    String messageString = message.getText();

    // Make sure that there is something to send!
```

```
        if ((null != messageString) && (0 < messageString.length()))
        {
            // Create a new message object.
            Message messageObj = pipe.createMessage();

            // Send the message using the output pipe.
            messageObj.setString("ChatMessage", messageString);

            // Send the message.
            try
            {
                outputPipe.send(messageObj);
            }
            catch (IOException e2)
            {
                System.out.println("Error sending..." + e2);
            }

            // Update the user interface.
            conversationText.append("\n");
            conversationText.append(displayName).append("> ");
            conversationText.append(messageString);
            conversation.setText(conversationText.toString());
            message.setText("");
        }
    }
}
```

The `ChatDialog` user interface enables a user to send a message to the remote user and view a history of the chat session. The `ChatDialog` class is also responsible for managing the input pipe used to send messages and the output pipe used to receive messages.

The Main Application

Like the `ExampleServiceTest` class in Chapter 10, the `JxtaMessenger` class shown in Listing 11.19 is responsible for generating the necessary advertisements for the Chat and Presence services and the `BuddyList` application, and for creating the peer group.

Listing 11.19 Source Code for *JxtaMessenger.java*

```
package com.newriders.jxta.chapter11;

import java.util.Hashtable;

import java.net.MalformedURLException;
import java.net.UnknownServiceException;
import java.net.URL;

import net.jxta.discovery.DiscoveryService;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;

import net.jxta.exception.PeerGroupException;

import net.jxta.id.IDFactory;

import net.jxta.impl.peergroup.StdPeerGroupParamAdv;

import net.jxta.impl.protocol.EndpointAdv;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.peergroup.PeerGroupFactory;

import net.jxta.platform.ModuleClassID;
import net.jxta.platform.ModuleSpecID;
```

```
import net.jxta.protocol.ModuleClassAdvertisement;
import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.protocol.ModuleSpecAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;

import com.newriders.jxta.chapter11.chat.ChatService;

import com.newriders.jxta.chapter11.impl.chat.ChatServiceImpl;

import com.newriders.jxta.chapter11.impl.presence.PresenceServiceImpl;

import com.newriders.jxta.chapter11.impl.protocol.PresenceAdv;

import com.newriders.jxta.chapter11.presence.PresenceService;

import com.newriders.jxta.chapter11.protocol.PresenceAdvertisement;


/**
 * The main class responsible for creating the chat peer group and starting
 * the Chat and Presence services and the BuddyList application.
 */
public class JxtaMessenger
{
    /**
     * The Peer Group ID for the chat group.
     */
    private static final String refPeerGroupID =
        "urn:jxta:uuid-68B8A7A691684F9C9E05971D66D78ED602";

    /**
     * The Module Specification ID for the peer group's Module
     * Implementation Advertisement.
     */
    private static final String refPeerGroupSpec =
```

```
    "urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000019E2"
    + "DDBBCE5FE4957A139B0ECE8DEB46D06";

/**
 * The new group created by the application.
 */
private PeerGroup newGroup = null;

/**
 * The Net Peer Group for the application.
 */
private PeerGroup netPeerGroup = null;

/**
 * Create the main application class.
 */
public JxtaMessenger()
{
    super();
}

/**
 * Creates a Module Class Advertisement using the given parameters.
 *
 * @param      moduleClassID the Module Class ID for the advertisement.
 * @param      name the symbolic name of the advertisement.
 * @param      description the description of the advertisement.
 * @exception  UnknownServiceException if the moduleClassID string
 *             is malformed.
 * @exception  MalformedURLException if the moduleClassID string
 *             is malformed.
 */
private ModuleClassAdvertisement createModuleClassAdv(
    String moduleClassID, String name, String description)
        throws UnknownServiceException, MalformedURLException
{
```

```
    // Create the class ID from the refModuleClassID string.
    ModuleClassID classID = (ModuleClassID) IDFactory.fromURL(
        new URL((moduleClassID)));

    // Create the Module Class Advertisement.
    ModuleClassAdvertisement moduleClassAdv =
        (ModuleClassAdvertisement)
            AdvertisementFactory.newAdvertisement(
                ModuleClassAdvertisement.getAdvertisementType());

    // Configure the Module Class Advertisement.
    moduleClassAdv.setDescription(description);
    moduleClassAdv.setModuleClassID(classID);
    moduleClassAdv.setName(name);

    // Return the advertisement to the caller.
    return moduleClassAdv;
}

/**
 * Creates a Module Implementation Advertisement for the service using
 * the specification ID in the passed in ModuleSpecAdvertisement
 * advertisement.  Use the given ModuleImplAdvertisement to create the
 * compatibility element of the Module Implementation Advertisement.
 *
 * @param       groupImpl the ModuleImplAdvertisement of the parent
 *              peer group.
 * @param       moduleSpecAdv the source of the specification ID.
 * @param       description of the module implementation.
 * @param       code the fully qualified name of the module
 *              implementation's class.
 * @return      the generated Module Implementation Advertisement.
 */
private ModuleImplAdvertisement createModuleImplAdv(
    ModuleImplAdvertisement groupImpl,
    ModuleSpecAdvertisement moduleSpecAdv,
```

```
        String description, String code)
    {
        // Get the specification ID from the passed advertisement.
        ModuleSpecID specID = moduleSpecAdv.getModuleSpecID();

        // Create the Module Implementation Advertisement.
        ModuleImplAdvertisement moduleImplAdv =
            (ModuleImplAdvertisement)
AdvertisementFactory.newAdvertisement(
                ModuleImplAdvertisement.getAdvertisementType());

        // Configure the Module Implementation Advertisement.
        moduleImplAdv.setCode(code);
        moduleImplAdv.setCompat(groupImpl.getCompat());
        moduleImplAdv.setDescription(description);
        moduleImplAdv.setModuleSpecID(specID);
        moduleImplAdv.setProvider("Brendon J. Wilson");

        // Return the advertisement to the caller.
        return moduleImplAdv;
    }

    /**
     * Creates a Module Specification Advertisement using the
     * given parameters.
     *
     * @param       moduleSpecID the Module Specification ID for
     *              the advertisement.
     * @param       name the symbolic name of the advertisement.
     * @param       description the description of the advertisement.
     * @exception   UnknownServiceException if the moduleSpecID string
     *              is malformed.
     * @exception   MalformedURLException if the moduleSpecID string
     *              is malformed.
     */
```

```
    private ModuleSpecAdvertisement createModuleSpecAdv(String
moduleSpecID,
        String name, String description)
            throws UnknownServiceException, MalformedURLException
    {
        // Create the specification ID from the refModuleSpecID string.
        ModuleSpecID specID = (ModuleSpecID) IDFactory.fromURL(
            new URL((moduleSpecID)));

        // Create the Module Specification Advertisement.
        ModuleSpecAdvertisement moduleSpecAdv =
            (ModuleSpecAdvertisement)
AdvertisementFactory.newAdvertisement(
                ModuleSpecAdvertisement.getAdvertisementType());

        // Configure the Module Specification Advertisement.
        moduleSpecAdv.setCreator("Brendon J. Wilson");
        moduleSpecAdv.setModuleSpecID(specID);
        moduleSpecAdv.setDescription(description);
        moduleSpecAdv.setName(name);
        moduleSpecAdv.setSpecURI(
            "http://www.brendonwilson.com/projects/jxta");
        moduleSpecAdv.setVersion("1.0");

        // Return the advertisement to the caller.
        return moduleSpecAdv;
    }

    /**
     * Creates a peer group and configures the ChatService and
     * PresenceService implementations to run as peer group services,
     * and configures the BuddyList as a peer group application.
     *
     * @exception   Exception, PeerGroupException if there is a problem
     *              while creating the peer group or the service
     *              advertisements.
```

```
 */
public void createPeerGroup() throws Exception, PeerGroupException
{
    // The name and description for the peer group.
    String name = "JXTA Messenger Group";
    String description =
        "A peer group for the Chapter 11 example application.";

    // The Discovery service to use to publish the module and peer
    // group advertisements.
    DiscoveryService discovery = netPeerGroup.getDiscoveryService();

    ModuleImplAdvertisement implAdv =
        netPeerGroup.getAllPurposePeerGroupImplAdvertisement();

    // Create the module advertisements for the Presence service.
    ModuleClassAdvertisement presenceClassAdv = createModuleClassAdv(
        PresenceService.refModuleClassID, "Presence Service",
        "A service to provide presence information.");
    ModuleSpecAdvertisement presenceSpecAdv = createModuleSpecAdv(
        PresenceServiceImpl.refModuleSpecID, "Presence Service",
        " A Presence service specification");
    ModuleImplAdvertisement presenceImplAdv = createModuleImplAdv(
        implAdv, presenceSpecAdv,
        "The reference Presence service implementation",
        "com.newriders.jxta.chapter11.impl.presence."
        + "PresenceServiceImpl");

    // Create the module advertisements for the Chat service.
    ModuleClassAdvertisement chatClassAdv = createModuleClassAdv(
        ChatService.refModuleClassID, "Chat Service",
        "A service to provide chat capabilities.");
    ModuleSpecAdvertisement chatSpecAdv = createModuleSpecAdv(
        ChatServiceImpl.refModuleSpecID, "Chat Service",
        "A Chat service specification");
    ModuleImplAdvertisement chatImplAdv = createModuleImplAdv(
```

```
            implAdv, chatSpecAdv,
            "The reference Chat service implementation",
            "com.newriders.jxta.chapter11.impl.chat.ChatServiceImpl");

        // Create the module advertisements for the BuddyList application.
        ModuleClassAdvertisement appClassAdv = createModuleClassAdv(
            BuddyList.refModuleClassID , "BuddyList Application",
            "An application providing a simple chat application.");
        ModuleSpecAdvertisement appSpecAdv = createModuleSpecAdv(
            BuddyList.refModuleSpecID, "BuddyList Application",
            "The BuddyList application specification");
        ModuleImplAdvertisement appImplAdv = createModuleImplAdv(
            implAdv, appSpecAdv ,
            "The reference BuddyList application implementation",
            "com.newriders.jxta.chapter11.BuddyList");

        // Get the parameters for the peer group's Module Implementation
        // Advertisement to which we will add our service.
        StdPeerGroupParamAdv params =
            new StdPeerGroupParamAdv(implAdv.getParam());

        // Get the services from the parameters.
        Hashtable services = params.getServices();

        // Add the Chat and Presence services to the set of services.
        services.put(presenceClassAdv.getModuleClassID(),
presenceImplAdv);
        services.put(chatClassAdv.getModuleClassID(), chatImplAdv);

        // Set the services on the parameters.
        params.setServices(services);

        // Replace the applications in the parameters.
        Hashtable applications = new Hashtable();

        // Add the BuddyList to the applications.
```

```
applications.put(appClassAdv.getModuleClassID(), appImplAdv);

// Set the applications on the parameters.
params.setApps(applications);

// Set the parameters on the implementation advertisement.
implAdv.setParam((StructuredDocument) params.getDocument(
    new MimeMediaType("text", "xml")));

// VERY IMPORTANT! You must change the module specification ID for
// the implementation advertisement. If you don't, the new peer
// group's module specification ID will still point to the old
// specification, and the new service will not be loaded.
implAdv.setModuleSpecID((ModuleSpecID) IDFactory.fromURL(
    new URL(refPeerGroupSpec)));

// Publish the Presence module class and spec advertisements.
discovery.publish(presenceClassAdv, DiscoveryService.ADV);
discovery.remotePublish(presenceClassAdv, DiscoveryService.ADV);
discovery.publish(presenceSpecAdv, DiscoveryService.ADV);
discovery.remotePublish(presenceSpecAdv, DiscoveryService.ADV);

// Publish the Presence module class and spec advertisements.
discovery.publish(chatClassAdv, DiscoveryService.ADV);
discovery.remotePublish(chatClassAdv, DiscoveryService.ADV);
discovery.publish(chatSpecAdv, DiscoveryService.ADV);
discovery.remotePublish(chatSpecAdv, DiscoveryService.ADV);

// Publish the Peer Group implementation advertisement.
discovery.publish(implAdv, DiscoveryService.ADV);
discovery.remotePublish(implAdv, DiscoveryService.ADV);

// Create the Peer Group ID.
PeerGroupID groupID = (PeerGroupID) IDFactory.fromURL(
    new URL((refPeerGroupID)));;
```

```
    // Create the new group using the group ID, advertisement, name,
    // and description.
    newGroup = netPeerGroup.newGroup(groupID, implAdv, name,
        description);

    // Need to publish the group remotely only because newGroup()
    // handles publishing to the local peer.
    PeerGroupAdvertisement groupAdv =
        newGroup.getPeerGroupAdvertisement();
    discovery.remotePublish(groupAdv, DiscoveryService.GROUP);

    // Start the peer group's applications.
    newGroup.startApp(null);
}

/**
 * Starts the JXTA platform.
 *
 * @exception   PeerGroupException thrown if the platform can't
 *              be started.
 */
public void initializeJXTA() throws PeerGroupException
{
    netPeerGroup = PeerGroupFactory.newNetPeerGroup();
}

/**
 * Starts the application.
 * @param args an array of command-line arguments
 */
public static void main(String[] args)
{
    JxtaMessenger messenger = new JxtaMessenger();

    try
    {
```

```
            // Initialize the JXTA platform.
            messenger.initializeJXTA();

            // Create the group.
            messenger.createPeerGroup();
        }
        catch (Exception e)
        {
            System.out.println("Error starting JXTA platform: " + e);
            System.exit(1);
        }
    }
}
```

The `JxtaMessenger` 's `createPeerGroup` does the majority of the work, generating the Module Implementation Advertisement for the peer group and populating it with the Module Implementation Advertisements for each of the two services and the application. The required Module Class IDs and Module Specification IDs were generated using the `GenerateID` application developed in Chapter 10 and were stored as static variables in `ChatService`, `ChatServiceImpl`, `PresenceService`, `PresenceServiceImpl`, and `BuddyList` for convenience.

Running JXTA Messenger

After all the classes have been compiled, you should be able to run the JXTA Messenger application from a directory containing the JXTA JARs using this code:

```
java -classpath .;beepcore.jar;cms.jar;cryptix32.jar;cryptix-asn1.jar;
instantp2p.jar;jxta.jar;jxtaptls.jar;jxtasecurity.jar;jxtashell.jar;
log4j.jar;minimalBC.jar com.newriders.jxta.chapter11.JxtaMessenger
```

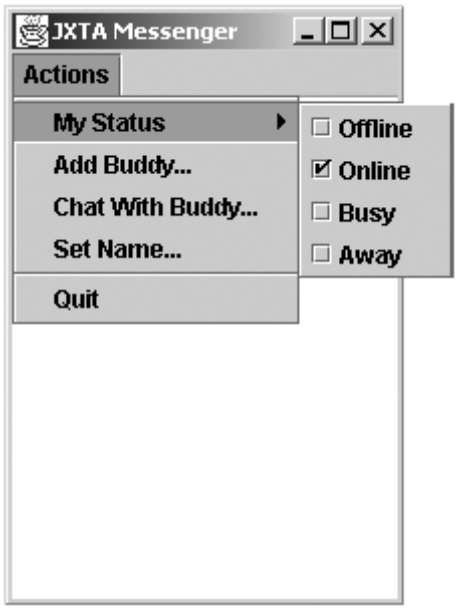
To see the JXTA Messenger in action, you will probably want to start a second instance of `JxtaMessenger` from a different directory using different TCP and HTTP ports. This will enable you to experiment with the application on your own machine.

After the peer group has been created and started by `JxtaMessenger`, the `BuddyList` application prompts you to enter an email address and display a name to use when sending presence information to other peers. This information will also be used when requesting or approving a chat session. After you have entered this information, the main user interface should appear as shown in Figure 11.1.

Figure 11.1. The main JXTA Messenger user interface

By default, your presence status will be set to Online.You can change your presence status from the My Status menu item under the Actions menu, as shown in Figure 11.2.

Figure 11.2. Setting your presence status.



When you set your status, a Presence Advertisement is published both locally and remotely using the peer group's Presence service.

To monitor other users' presence status, you need to add the user to your list of buddies using the Add Buddy menu item under the Actions menu.You will be prompted for an email address that the `BuddyList` application can use to search for a Presence Advertisement using the Presence service. The buddy will be added to the list, but the buddy's status will remain as Offline until `BuddyList` 's `presenceUpdated` method is called by the `PresenceService` instance to notify `BuddyList` that a Presence Advertisement has been found for the buddy.

When presence information is received, the `BuddyList` instance updates both the user interface and its internal set of buddies. The Peer ID associated with the buddy is also stored to allow the user to send chat requests to the buddy without using propagation.

To chat with a buddy, click a buddy in the list and select the Chat with Buddy menu item under the Actions menu. The `BuddyList` class uses the Chat service to send an Initiate Chat Request Message to the remote user. When the `BuddyList` class receives a request to initiate a chat session, it checks whether the requesting buddy is in the user's list of buddies. If it is, the `BuddyList` class creates a pipe using the `BidirectionalPipeService` and

sends an Initiate Chat Response Message. Otherwise, the `BuddyList` class ignores the request. This means that only users who are in each other's buddy list can chat.

When the `BuddyList` receives notification that the request to chat has been approved, it attempts to connect to the remote peer using the Pipe Advertisement contained in the response. If this connection is successful, the `BuddyList` spawns a `ChatDialog` to manage the chat session, as shown in Figure 11.3.

Figure 11.3. The `ChatDialog` user interface.

`ChatDialog` uses the `OutputPipe` bound by the `BidirectionalPipeService` to send messages entered by the user. Messages received on the `InputPipe` bound by the `BidirectionalPipeService` are added to the text area showing the conversation history. When the user closes the dialog box, the pipes are closed and no further communication is possible without requesting a chat session using the same procedure as before.

Summary

Although the chat application developed in this chapter might not be as fully featured as MSN Messenger or ICQ, it still demonstrates how the JXTA protocols covered in this book can be used to create a complete P2P solution:

- **The Peer Discovery Protocol—** The Presence service implementation uses the Discovery service to search for Presence Advertisements containing presence information for a user. The Discovery service is also used by the Presence service implementation to discover presence information for a user.

- **The Resolver Protocol—** The Chat service implementation uses the Resolver service to handle sending and receiving messages used to request and approve a request to establish a chat session.

- **The Rendezvous Protocol—** Although it is not used directly, the Rendezvous service provides the Discovery service with the capability to publish Presence Advertisements to many peers.

- **The Pipe Binding Protocol—** The Pipe Binding Protocol is used by the `BidirectionPipeService` 's use of the Pipe service to establish a two-way communication channel to a remote peer for conducting the chat session.

- **The Endpoint Routing Protocol—** Underneath all the protocols, the Endpoint Routing Protocol provides the mechanism required to route messages to destination peers.

The only protocol not used by this application is the Peer Information Protocol, which, given its current state, is forgivable. The sample application also re-enforces the demonstration of peer groups, modules, and services given in Chapter 10.

So where do you go from here? The next chapter examines some of the more ambitious projects currently being built by the JXTA Community. The next chapter also provides information on how you can get involved with Project JXTA, contribute to an existing project, or propose a project of your own.

Chapter 12. The Future of JXTA

Now that you've seen jxta, you're probably wondering what your next step should be. Maybe you want to help Project JXTA refine the current Java reference implementation, or work on a reference implementation in your preferred development language.You might have a new project that you want to tackle with this new tool, or you might just want to see what other people are working on. This chapter provides information on where JXTA is heading and how you can participate.

Future Directions for Project JXTA

At several points in this book, you might have gotten the impression that JXTA is not currently a "complete" product. In truth, it might never be complete. Like many open-source projects, JXTA is constantly evolving as open-source developers augment the existing reference implementation and specification to address new problems in peer-to-peer computing.

Besides working on the Java reference implementation, developers within the JXTA Community are developing new services and applications built on top of the JXTA platform. The future success of JXTA depends on the capability of these applications and services to demonstrate the benefits and viability of JXTA technology. In addition to working on applications of the JXTA technology, other developers are creating new reference implementations or the core JXTA platform in languages other than Java. These new reference implementations will enable developers in a variety of languages to recognize the benefits of JXTA in their preferred development language.

The following sections outline some of the most prominent services, applications, and new reference implementations being developed by the JXTA Community. This is by no means

a comprehensive list, and it does not include applications that are being developed outside the JXTA Community hosted by Project JXTA. For the latest, look at the list of projects hosted by Project JXTA at www.jxta.org/servlets/DomainProjects.

Services

Services supply the building blocks that applications can use to provide real functionality to an end user. Services can be independent or can augment other services to provide new functionality. Some services might provide mechanisms of their own that replace the core services provided in the reference implementation, such as the Discovery service.

Nearly two dozen services projects currently are listed in the Services category of projects hosted by the Project JXTA site. These projects are in many different stages of development. Some projects are just beginning to do their initial design work, while others are working on coding implementations. This section outlines three of the most developed services within the JXTA Community. More information on services being developed by the JXTA Community is available from the Project JXTA Services web site, services.jxta.org.

JXTA Search

The JXTA Search project began at Infrasearch, a company that was later acquired by Sun Microsystems and folded into Project JXTA. Infrasearch's proof-of-concept technology used the Gnutella protocol to allow search clients to query information providers, such as web sites, to gain access to information not available via traditional search engines.

The JXTA Search project is building on that initial technology using JXTA to enable information consumers to search for information locked away in the "deep web." As outlined in Chapter 1, "Introduction," current search engines are limited to indexing static information provided on the web, resulting in irrelevant or outdated responses. In addition, traditional search-engine technology does not capture or index information that is stored within corporate databases. As a result, search-engine query results are neither as comprehensive or as real-time as possible.

JXTA Search solves the problem by allowing information providers to integrate their corporate information stores with a distributed network of peers. The JXTA Search project defines a set of protocols built on top of JXTA that allows a client to query an information provider and obtain results from the information provider's store of information. The advantage of this approach is that the results of queries are more up-to-date and comprehensive than those obtained using search engines.

Note

JXTA Search doesn't necessarily replace current search-engine technology. Current search-engine technology is fairly well suited to the problem of indexing static pages. JXTA Search simply augments traditional search engines to provide access to information not captured by the indexing crawlers employed by search engines.

More information on the JXTA Search project can be found at the project's web site, search.jxta.org.

Content Management System (CMS)

The Content Management System (CMS) is designed to allow peers to share content with and retrieve shared content from other peers. The CMS service provides a foundation that other file- or document-sharing applications can use to handle the details of publishing and retrieving content from a set of distributed peers.

The CMS defines a Content Advertisement, which provides metadata describing a particular piece of content. The Content Advertisement itself includes an MD5 hash generated from the content data that uniquely identifies the content. This MD5 hash can be used by the peer to retrieve content from any peer hosting the content, without relying on parameters that could change, such as filename.

In addition to the advertisement definition, the CMS specifies a protocol for searching for and retrieving content. This protocol augments the basic JXTA pipe functionality to allow a peer to download content from a peer and ensure that the content is retrieved in a reliable fashion. Content shared by a peer is managed by the CMS in a persistent store containing the shared content and the content advertisements.

In the future, the CMS project hopes to augment the current search functionality to incorporate the JXTA Search service. More information on the project's progress can be found at the project's web site, cms.jxta.org.

JXTA-Remote Method Invocation (RMI)

The Remote Method Invocation API provided by the Java 2 Standard Edition allows a program to invoke methods on Java objects hosted on another JVM or even on a remote machine. By default, RMI uses TCP/IP as a network transport to invoke methods on remote objects and transports serialized object instances.

The JXTA-RMI service enables a Java developer to use JXTA pipes instead of TCP/IP as a transport mechanism. The advantage of this approach is that it allows existing RMI-based applications to build on JXTA without requiring major changes to the existing application. This implementation still provides only a point-to-point solution, despite JXTA's capability to handle many-to-many communication.

More information on RMI is available from Sun at java.sun.com/products/jdk/rmi/index.html, and current project information and source code for JXTA-RMI is available from the project's web site, jxta-rmi.jxta.org.

Applications

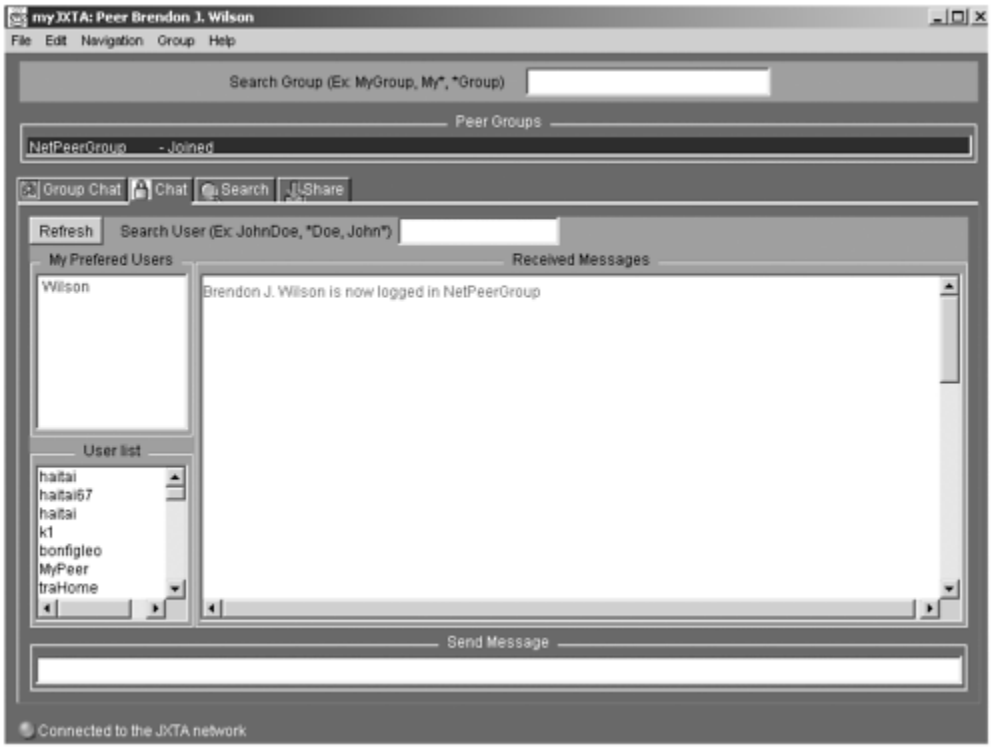
Applications build on existing JXTA services to provide an end user with some useful way of interacting with the JXTA P2P network. Usually the application incorporates some form of user interface, enables users to interact with JXTA services, and controls their behavior.

Currently more than a dozen application projects are listed in the Applications category of projects hosted by the Project JXTA site. One application being developed, the JXTA Shell, formed the basis of the examples in the first half of the book. This section outlines two of the most developed applications within the JXTA Community. More information on applications being developed by the JXTA Community is available from the Project JXTA Applications web site, apps.jxta.org.

myJXTA

Besides the JXTA Shell, myJXTA, shown in Figure 12.1, is probably one of the most fully featured JXTA applications. Originally called InstantP2P, myJXTA was one of the first applications developed to demonstrate the capabilities of the JXTA platform. This application is currently designed to run only on desktop computers, but it should eventually be capable of running on devices supporting the PersonalJava or J2ME platforms.

Figure 12.1. The myJXTA user interface.



The myJXTA application enables a user to participate in a group chat room, engage in one-to-one chat, share files, and create, join, and leave peer groups. The file-sharing functionality of myJXTA is built on top of the Content Management Service discussed in the previous section.

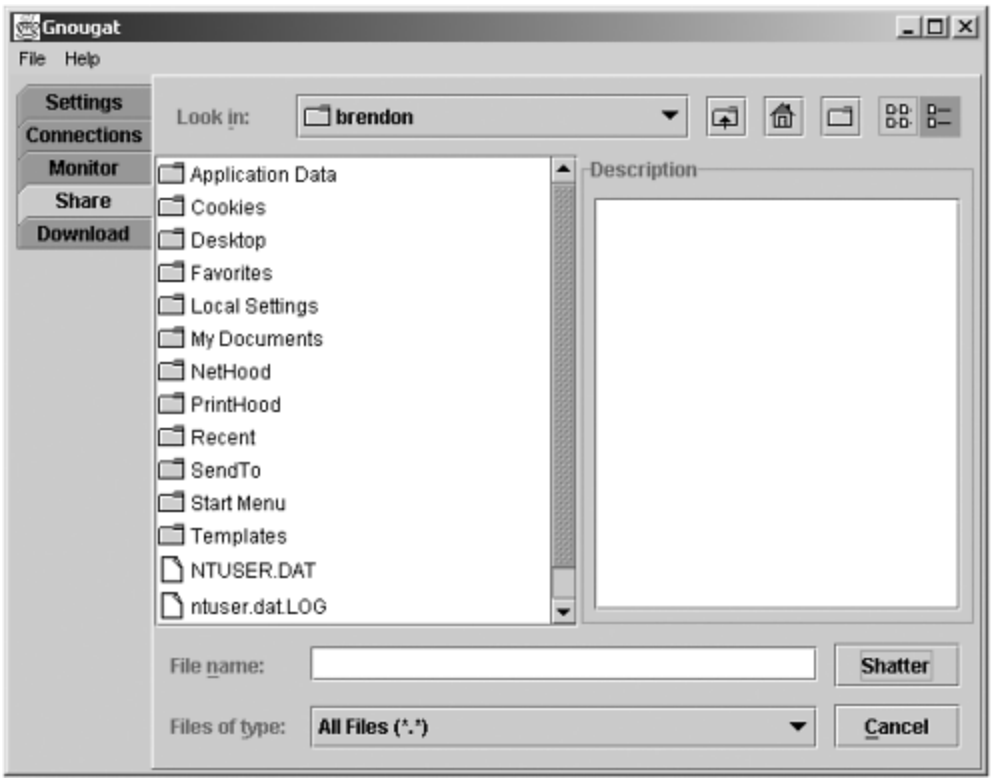
Because the myJXTA application is fairly stable, future plans include only fixing known bugs. The myJXTA source code is a good place to start when changes are made to the JXTA reference API because an updated version of the application is usually released at roughly the same time as a new stable release of the Java reference implementation. More

information on the project and its source code can be found at the project's web site, instantp2p.jxta.org.

Gnougat

Gnougat, shown in Figure 12.2, is a file-sharing application similar to Gnutella, with the notable exception that the responsibilities of file sharing can be distributed across all peers in the network.

Figure 12.2. The Gnougat user interface.



In traditional file-sharing applications, peers have no way to determine whether two search results point to the same content. Searches are performed based on the name of a shared file or metadata embedded in the file. Unfortunately, there is no guarantee that two files with matching metadata are identical. Gnougat attempts to solve this problem by focusing on content instead of metadata information.

In the Gnougat system, a file containing specific content is downloaded using a hash of the content itself. This allows the client peer to find peers hosting the same content and optimize the download of the file. In addition, the system allows client peers to split the download into multiple chunks, each of which can be downloaded in parallel from a different peer.

The Gnougat application still uses metadata to search for shared files. Content is discovered initially using metadata, such as filename; then a hash descriptor for a particular piece of content is retrieved from the network. This hash descriptor is used by the client peer to perform the download by discovering peers hosting the exact same content.

A preliminary implementation of Gnougat has been made available by the developers. The installer, along with a discussion paper on the technology behind Gnougat, is available from the project's web site, gnougat.jxta.org.

Core Reference Implementations

A variety of non-Java reference implementations currently are in progress within the JXTA Community. Projects currently have been established for the C, Objective C, Perl, and Ruby languages.

Other implementation projects are focusing on producing JXTA implementations for small or constrained devices:

- The JXME project is producing a version of the JXTA platform suitable for handheld devices capable of running the Java 2 Micro Edition platform.
- The PocketPC project is also porting the JXTA reference implementation to C++ specifically for the PocketPC and CE platforms.
- The TINI project is working on a version of JXTA for the TINI platform, an embedded Java runtime designed to run on top of the DS80C390 microprocessor from Dallas Semiconductor.

At this point, most of these projects are in the very early stages of design and implementation. The Java reference implementation remains the most complete implementation of the JXTA specification at this time.

Participating in Project JXTA

The future of JXTA depends on developers contributing their skills to Project JXTA. As previously mentioned, JXTA is currently a work-in-progress, one that requires people to help with all aspects of the development. Whether you're interested in starting a new JXTA project or helping out with existing projects, Project JXTA will benefit from your involvement.

To start getting involved with Project JXTA, you should first do the following:

1. **Register.** If you are interested in contributing to an existing project or starting a project of your own, you should first become a member of the JXTA Community. Registration is free and can be done from the Project JXTA registration page at www.jxta.org/servlets/Join. Registering enables you to join projects as a contributor and propose projects of your own.

2. **Submit a contributor agreement.** To contribute code to Project JXTA, you must submit a contributor agreement. The agreement is available for download from www.jxta.org/jxta_contrib_agreement.PDF. This agreement ensures that the code you contribute is legally yours to contribute and is being contributed in accordance with the license employed by Project JXTA.

3. **Join the mailing lists.** The JXTA mailing lists are used to propose new projects, analyze problems to be tackled by peer-to-peer, and announce changes in the reference implementation.You should join at least the discuss@jxta.org mailing list to keep abreast of the latest JXTA developments.

The JXTA Mailing Lists

Project JXTA maintains a number of mailing lists to allow people to discuss aspects of JXTA and peer-to-peer technology. The four main mailing lists hosted by Project JXTA are listed here:

- **announce@jxta.org**— A list used to announce major releases to members of the general public who are interested in JXTA. This list is not meant to host discussions.

- **vdev@jxta.org**— A list for developers working with JXTA to discuss technical issues related to JXTA. Currently, most of the discussion on this list focuses on the Java reference implementation of JXTA.

- **discuss@jxta.org**— A more general discussion list devoted to JXTA and the problems that need to be addressed by peer-to-peer systems. This list is also used to propose new projects to the JXTA Community and seek approval from the community.

- **user@jxta.org**— A list for new JXTA developers who are just starting to familiarize themselves with the reference implementation. Common themes include questions on using the JXTA Shell, using myJXTA, and performing basic operations with the reference implementation APIs.

Instructions for subscribing and unsubscribing from the mailing lists, as well as searchable archives of the lists, are available from www.jxta.org/project/www/maillist.html. As with any mailing list, you should monitor the mailing list for a short time before posting new messages, to ensure that you are posting content that is appropriate for the mailing list.

In addition to these four lists, each project hosted by Project JXTA hosts its own mailing lists. By default, each project hosts a `cvs` mailing list that announces when changes are committed to the project's source control and an `issues` mailing list that announces messages from the project's bug-tracking system. Although projects are free to create other discussion mailing lists, Project JXTA encourages projects to use the main `dev` and `discuss` mailing lists unless the project generates an unusual amount of mail. This ensures that the JXTA Community is apprised of developments in all projects currently under development.

Proposing a New JXTA Community Project

All this talk about the advantages of JXTA might have you thinking of developing a killer peer-to-peer application of your own. But before you run out and start development, consider proposing the project to the JXTA Community for any of the following reasons:

- **To obtain constructive criticism—** A lot of smart people are involved in the JXTA Community, and chances are good that some of them have considered the problem that your application or service will attempt to solve. Proposing the project to the community will allow discussion of the potential roadblocks that you will encounter and generate valuable feedback.

- **To eliminate duplication—** There's no point in starting a whole new project if there's currently a similar effort under way within the JXTA Community. Proposing the project to the community ensures that your application or service isn't a duplication or variation of an existing project. If there's a matching project in existence, you benefit by being able to join a team that has already done some of the work.

- **To form a team—** You don't want to do all this work yourself, do you? Proposing the project to the community gives you a chance to gather interested developers to help you implement your application.

- **To take advantage of tool hosting—** If your project is approved, you will be able to manage the project using the tools provided by jxta.org. These tools include hosted source control, bug tracking, and project mailing lists. These tools not only relieve you of the burden of hosting these tools yourself, but they also allow others to easily access and contribute to your project.

To propose a new project to the JXTA Community, you should first register with Project JXTA, submit a contributor's agreement, and subscribe to the discuss@jxta.org mailing list. After you have done this, send a message to the discuss@jxta.org mailing list with the project's proposal. The project proposal should contain the following:

- A descriptive subject line that includes the text "Proposed Project:" and the name of the proposed project.

- A complete description of the project and the particular problem that the project will attempt to solve.

- A description of the project category. The categories correspond to those on the projects page: Apps, Core, Demos, Other, or Services.

You must ensure that your project will conform to the terms of the license employed by Project JXTA. License information is available from www.jxta.org/license.html.

After you have posted a proposal message to the mailing list, you should begin receiving feedback indicating whether the JXTA Community thinks that your project would be worth undertaking. While this discussion is occurring, you should also register with Project JXTA, create the project's home page, and formally propose the project to the jxta.org Community Manager using the form at www.jxta.org/servlets/ProjectAddStep1. If your proposal is complete and the discussion on the mailing list indicates that there is reason to pursue the proposed project, the Community Manager should approve your project after a few days.

Working with the Java Reference Implementation Source Code

The Java reference implementation is the most active project currently under way in the JXTA Community, and many other projects build on its capabilities. Because most of the Java-based projects have a common build system, this section shows you how to obtain the latest version of the reference implementation and build it.

Obtaining Java Reference Implementation Source Code

As previously mentioned, Project JXTA provides hosted projects with a source-control system to use to manage the project's source code. The source-control system used by Project JXTA is the Concurrent Versions System (CVS), which allows multiple developers to work on the same code simultaneously and merge their work.

To access the CVS source repository, you'll need to install a CVS client. For the demonstration, I'll be using WinCVS, a Windows-based CVS client available from www.wincvs.com that provides a simple user interface. Non-Windows and command-line

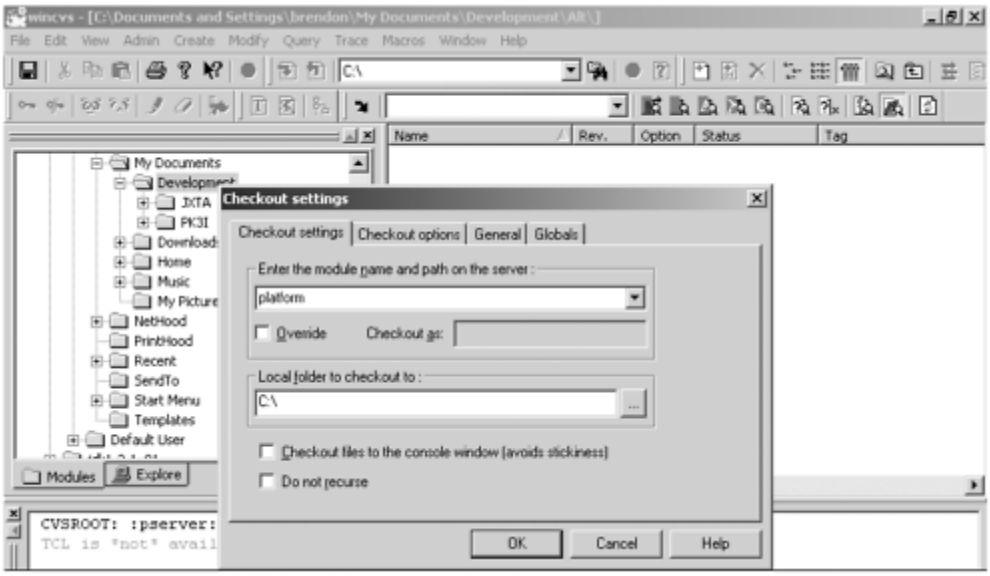
CVS clients are available from www.cvshome.org/downloads.html. Manuals and information on CVS are also available from www.cvshome.org.

To obtain the Java reference implementation, follow these steps:

1. Start WinCVS.

2. Select the Checkout Module item under the Create menu. The Checkout Settings dialog box displays (see Figure 12.3).

Figure 12.3. The Checkout Settings dialog box.



3. Enter **platform** in the Checkout Settings dialog box. This is the CVS module name of the Java reference implementation in the source code repository.

4. Select the local folder where the retrieved source code will be placed.

5. Go to the General tab.

6. Enter **:pserver:guest@cvs.jxta.org:/cvs** into the text field for the CVSROOT. This specifies the user to use to retrieve the source code.

   For this example, just use the guest user.

7. Ensure that Authentication is set to Password File on the CVS Server.

8. Click OK.

The CVS client should now contact the server, retrieve the source code for the Java reference implementation, and place it in the specified local directory. When the CVS client has completed the download from source control, you can build the downloaded source code.

Building the Java Reference Implementation Source Code

The Java reference implementation, as well as many of the Java-based JXTA Community projects, use the Ant build tool created by the Apache Jakarta project (jakarta.apache.org). To build the reference implementation, you first should download and install the latest version of the Ant build tool from the Ant home page at jakarta.apache.org/ant/index.html.

Ant provides a build tool similar to `make`, but it is based on Java and is therefore cross-platform. To use Ant, you need only to install Ant and add the path to Ant's `bin` to your system path. On some systems, you might also need to set a `JAVA_HOME` variable to point to the location of the JDK. To build the reference implementation, follow these steps:

1. Open a command prompt.

2. Change to the directory containing the reference implementation code that you retrieved from source control.

3. Change to the `binding\java` subdirectory.

4. Set the `JAVA_HOME` variable. On Windows, this is achieved by typing **SET JAVA_HOME=** followed by the fully qualified path to your JDK (for example, `C:\jdk1.3.1_01\`).

5. Include the Ant `bin` directory in your system path. On Windows, this is achieved by typing **SET PATH=%PATH% ;** followed by the fully qualified path to the Ant `bin` directory.

6. Build the source code by typing **ant**.

The Ant build tool will use the `build.xml` file to compile all the Java reference implementation source code and create the `jxta.jar` file. The compiled classes will be placed in the `classes` subdirectory, and `jxta.jar` will be placed in the `lib` subdirectory.

Summary

In this chapter, you learned about a few of the services, applications, and implementations of the JXTA technology, and you learned how to get involved with Project JXTA to help shape the future of the technology. The rest, to paraphrase my university professors, "is left as an exercise for the reader."

This book has been about teaching people how to use JXTA so that they can go out and define the direction of peer-to-peer technology. I hope that this book has helped give you the tools that you need to go out and produce your own solutions. If you think that something should be included or expanded in future versions of the book (assuming that there is a second edition), feel free to drop me an email via my web site, www.brendonwilson.com.

Part IV: Appendixes

Appendix A. Glossary

Glossary

advertisement

A description of a resource made available on a P2P network. Types of resources described by advertisements include peers, peer groups, pipes, endpoints, content, and services. In JXTA, advertisements are represented using XML documents.

content

A generic term used to describe any type of text or binary data that can be stored and retrieved at a later time.

CVS

Concurrent Versions System, an open-source version-control system that allows multiple developers to work on source files simultaneously. CVS tracks changes made to source code files and allows developers to merge changes back into a version-control system, to provide an accurate, trackable, and reproducible record of a source file at any point in its development history.

endpoint

An abstraction of a peer's underlying native network interfaces. The term is also used to describe the source peer sending a message or the destination peer receiving a message through the P2P network.

HTML

Hypertext Markup Language, a set of text tags that are used to mark up plain text to provide formatting for a web browser. The HTML standard is currently migrating toward an XML-compati-ble schema.

HTTP

Hypertext Transfer Protocol, a protocol used to transfer HTML and web content. HTTP is a stateless protocol, which means that each request is completely independent of the requests that preceded it.

IETF

Internet Engineering Task Force, a standards body responsible for defining standard protocols for the Internet. Standards are developed by the IETF with the participation of the Internet community, and they are documented in Requests for Comments (RFCs).

IM

Instant messaging, a form of electronic communication that allows two or more parties to exchange text messages instantly. Examples of instant-messaging applications include ICQ, MSN Messenger, AOL Instant Messenger, and Yahoo! Messenger.

IP

Internet Protocol, a protocol used to send data from a source computer to a destination computer. IP is a low-level protocol that is responsible for sending packets of data across a network, possibly using a variety of routes to a destination. Packets sent across the network are treated independently, defining IP as a connectionless protocol. Both the source and the destination for the packet are

uniquely described in the packet's headers by an IP address. Unlike TCP, IP is not a reliable protocol and does not guarantee packet delivery.

JAR

Java Archive, a compressed binary archive format used to distribute compiled Java class files and resource files for an application.

Jini

A network technology from Sun that allows devices to spontaneously join networks and make their services available to other devices. Although there are some superficial similarities between Jini and JXTA, Jini is heavily dependent on the Java language; JXTA is designed to enable a developer to create a JXTA-compatible application independent of a particular operating system or programming language.

JRE

Java Runtime Environment, an environment used to run Java applications. The JRE consists of both a Java Virtual Machine and the standard Java runtime libraries.

JVM

Java Virtual Machine, a simulated computer environment used to run Java applications. Java is different from traditional computer languages in that source code is compiled into byte code rather than machine code for a particular processor. A JVM provides a mechanism for translating Java byte code into native machine code at execution time, allowing code to run on any platform that has a JVM implementation.

LAN

Local area network, a data-communications network connecting a set of computers in a limited local geographic area.

MD5

Message Digest 5, a hashing algorithm developed by Dr. Ronald Rivest of RSA Security. Hashing algorithms are mathematical one-way functions that convert a stream of bytes into a unique set of bytes, called a *message digest*. An important property of a good hashing algorithm is it makes it extremely difficult to construct two streams of bytes that result in the same message digest. Message digests are a fixed length, which makes them suitable for uniquely identifying a set of bytes or providing an integrity checksum.

metadata

Data that describes other data. In XML, the tags used to mark up data provide metadata that describes the type of data contained by the tag. Metadata provides a higher level representation of information and provides context for the data it describes or contains.

MIME

Multipurpose Internet Mail Extensions, a mechanism to allow the exchange of non-ASCII data via Internet mail. MIME defines not only a format for message data, but also MIME types that identify the type of data contained in a MIME message. MIME types are now used by a variety of applications besides email, including HTTP, to allow applications to identify and handle different types of data.

MP3

MPEG-1 Audio Layer 3, an audio compression format defined by the Motion Picture Experts Group.

NAT

Network Address Translation, a translation scheme used to protect private internal networks from unauthorized incoming connections. NAT, usually implemented by a private network's router, translates a private internal IP address to an external public IP address and stores the mapping in the router's translation table. Incoming connections undergo an inverse mapping procedure; if no mapping exists, the connection is blocked from entering the internal network.

P2P

Peer-to-peer, a networking paradigm that enables intermittently connected devices, usually separated from the public network by a firewall, to offer resources to and consume resources from other devices on the network using a common set of communication protocols.

packet

A unit of data used in network communications. Typically, a message being sent from one computer to another using IP is divided into a set of packets that are sent across the network in an independent manner to a destination where the message is reassembled.

PDA

Personal digital assistant, a handheld computing device that provides some combination of personal information management and communication/networking functionality. Examples of popular PDAs include the Palm Pilot, Handspring Visor, Samsung Yopy, and Compaq iPaq.

peer

An entity on the P2P network used to provide access to the resources of the node and consume resources from other entities on the network.

peer group

Peers on a P2P network that join together to serve a common purpose. Peer groups allow peers to segment the network space by application, security, and monitoring requirements.

pipe

A virtual communications channel that connects a source endpoint to one or more destination endpoints to permit message exchange.

rendezvous peer

A peer that provides simple peers with a way of discovering other peers and advertisements on the P2P network. A rendezvous peer also provides simple peers with the capability to propagate messages within a group, across boundaries between public and private networks. Some rendezvous peers also cache advertisements to reduce network traffic and improve efficiency.

router peer

A peer providing routing services to enable peers inside private internal networks behind firewall and NAT equipment to participate in a P2P network.

service

A mechanism for providing access to a resource over a network to other peers on a P2P network.

SGML

Standard Generalized Markup Language, a method of defining a document language that can be used to mark up documents with metadata using a set of tags. HTML is an SGML-based document language that defines a set of tags used to mark up documents for presenting within a web browser.

simple peer

The simplest type of peer on a P2P network. A simple peer provides resource to and consumes resources from other peers on the network. A simple peer is not responsible for forwarding messages on behalf of other peers or providing third-party information to the network.

SMTP

Simple Mail Transport Protocol, a protocol used for exchanging email between servers.

TCP

Transport Control Protocol, a protocol that defines rules to guarantee that the packets that form a message arrive at the destination in a timely fashion and are reassembled correctly. TCP is used in conjunction with IP, which provides the function of communicating packets across the network from a source to a destination, in a form called TCP/IP.

TTL

Time To Live, a property used to limit the propagation of messages between rendezvous peers. The TTL property of a message defines the maximum number of times that a message should be propagated to other peers. When a rendezvous peer receives a message, it decrements the message's TTL value. If the result is `0`, the message is not propagated to other peers; otherwise, the message is propagated using the new TTL value.

UDP

User Datagram Packet, a protocol that defines port numbers used to distinguish communication layered on top of IP and checksums to verify data integrity. UDP is used in conjunction with IP, which provides the actual function of communicating packets across the network from a source to a destination. Like IP, UDP is an unreliable protocol that does not guarantee delivery.

UTF-8

Unicode Transformation Format, an encoding scheme used to represent Unicode strings that is specifically optimized for representing ASCII characters.

WSDL

Web Services Description Language, an XML-based language used to define the services offered by a server and how to engage those services.

XML

eXtensible Markup Language, a language used to define a set of tags that can be used to structure textual data. Tags defined by an XML Document Type Definition are used to mark up data, thereby providing additional information, or metadata, about the data. XML is a simple text-based language that can be easily transformed into other formats using eXtensible Stylesheet Language Transform (XSLT).

Appendix B. Online Resources

No book can hope to cover every aspect of P2P, either now or in the future. However, a number of online resources can help any budding P2P developer become a guru in his own right. This section lists some of the most relevant online resources devoted to the P2P movement.

P2P Companies and Organizations

These companies, organizations, and applications are at the bleeding edge of the P2P revolution; in some cases, they have helped it achieve the foothold that it has in the world of modern computing.

Napster

www.napster.com

The bad boy of the P2P networking world, Napster is credited with starting the P2P revolution with its hybrid P2P MP3 file-sharing software. Napster uses a hybrid P2P network consisting of a centralized server that handles indexing of a user's song files and allowing peers to locate song files to download. Individual peers handle the file-transfer process independent of the centralized server. Unfortunately, the centralized portion of Napster's service ultimately made it vulnerable to legal attacks by the recording industry. Napster's copyright infringement–enabling software got it into hot water with the Recording Industry Association of America. However, it struggled on and is now set to relaunch with a new subscription service to provide access to licensed music using the same distribution technology.

Gnutella

www.gnutelliums.com

Napster's successor, Gnutella, adopted a pure P2P file-sharing model, allowing it to avoid the legal difficulties encountered by Napster. With no central server providing services, the Gnutella software provided no target for the recording industry to attack. Unlike Napster, Gnutella allows sharing of any type of file, not just MP3s. Although there is no longer a Gnutella program, dozens of clones have picked up where the original Gnutella application finished, improving the capabilities and performance of the file-sharing technology.

MojoNation

www.mojonation.net

One of the many niche content-distribution technologies built on P2P, MojoNation attempts to address the problems of P2P using an artificial currency, called *Mojo*. Users of the system earn Mojo by providing content, and they spend Mojo to gain access to content. This currency helps enforce resource sharing by participants in the P2P network and prevents the network from suffering the Tragedy of the Commons described in Chapter 1, "Introduction." MojoNation also breaks files into fragments, scattering them across the network via multiple download sites in parallel, to enable faster downloads from peers using dial-up.

Freenet

freenet.sourceforge.net

Another of the niche content-distribution P2P networks, Freenet provides anonymous and decentralized content distribution. Freenet uses strong cryptography to protect resources distributed across the network, making it almost impossible for an attacker to destroy information on the network by preventing peers from determining what information is stored in their local cache. The Freenet solution also mirrors high-demand content to multiple locations to provide efficient service and bandwidth usage across the network.

Groove Networks

www.groove.net

The brainchild of Ray Ozzie, the creator of Lotus Notes, Groove Networks is building a platform for providing services in *shared spaces* that allow users to form peer groups and interact directly. One of Groove's unique features is its capability to handle offline interaction: Changes made to a user's shared space are reflected in other members of the peer group. Groove aggregates common P2P applications to provide a complete solution for business interaction that includes instant messaging, file sharing, and group activities, such as document editing or whiteboarding. Currently the Groove client is supported only on Microsoft Windows platforms.

Jabber

www.jabber.com

Jabber produces an instant-messaging client and server capable of interacting with all the major IM services, including AOL Instant Messenger, MSN Messenger,Yahoo! Messenger, and ICQ. The Jabber client uses an XML-based protocol to interact with major IM services via the Jabber server. The company is attempting to provide a common platform for instant-messaging solutions, working closely with the Internet Engineering Task Force's IM standardization effort.

IAM Consulting's JXTA Development Toolkit

www.iamx.com/jxtaDevTools/index.htm

IAM Consulting is a consulting firm providing JXTA and Java expertise that has been working on a set of tools to simplify JXTA development. Its current toolkit includes a Peer Group Monitoring tool to allow a developer to view the activity of a peer group's Discovery service, as well as a message monitor to enable a developer to debug pipes. Members of IAM consulting are also responsible for starting the JXTA Special Interest Group (SIG) in New York. For more information on the JXTA SIG, see www.jxtasig.org.

P2P Magazines

These online magazines provide access to articles on a variety of P2P-related topics, including tutorials on developing P2P solutions, discussion of the future direction of P2P, and information on emerging P2P technologies.

OpenP2P

www.openP2P.com

OpenP2P's online articles discuss P2P and its implications from a technological, legal, and social point of view. With insightful writing, OpenP2P is usually on top of the latest P2P developments and offers a good starting point for learning about new P2P technologies.

IBM's DeveloperWorks, Java Section

www.ibm.com/developerworks/java

The Java section of IBM's DeveloperWorks site provides developers free access to a huge library of Java code and Java tutorials, which is an invaluable Java resource even if you're not working on P2P software. If you search the site, you'll find recent articles on Java and P2P, and Project JXTA.

Peer-To-Peer Central

www.peertopeercentral.com

Peer-To-Peer Central provides free articles on the development of P2P technology, reviews of new P2P development platforms, and industry perspectives on the importance of P2P. The site also enables users to purchase analysis papers and case studies on the P2P industry.

Project JXTA Resources

Project JXTA houses a variety of web sites devoted to specific aspects of the JXTA Community's development efforts. All these sites are accessible from the main Project JXTA web site, www.jxta.org, so only the most relevant sites are listed in this section.

JXTA Protocol Specifications

spec.jxta.org

The JXTA Protocols Specification project is responsible for maintaining the JXTA protocol documentation and ensuring that JXTA implementations are compliant with the specification. This site houses the most up-to-date version of the JXTA Protocols Specification in DocBook and HTML formats.

Project JXTA Downloads

download.jxta.org

The download site provides access to the latest JXTA binaries, including the latest stable JAR files for the JXTA Demo applications, daily builds, and a set of easy installers for the Demo applications.

JXTA Community Projects

www.jxta.org/servlets/DomainProjects

All the JXTA Community projects currently under way are accessible from this site, including projects creating applications, services, and core layer technology, as well as tutorials on the JXTA platform.

Internet Standards and Standards Bodies

Many of the technologies used by JXTA or related to JXTA are managed by a standards body. This section lists the standards and standards bodies most relevant to the JXTA platform.

The World Wide Web Consortium (W3C)

www.w3.org

The World Wide Web Consortium (W3C) is responsible for maintaining many of the popular standards used by Internet applications, including the XML, SOAP, HTTP, and HTML standards.

The Peer-to-Peer Working Group

www.p2pwg.com

The P2P Working Group is a consortium of P2P-related companies working to establish best-known practices for P2P solutions to provide an infrastructure for P2P computing. Although the web site doesn't house much content now, given the impressive list of member companies—including Intel, Groove Networks, United Devices, and many others—this working group likely will publish a wide variety of content devoted to P2P in the near future.

The XML 1.0 Standard

www.w3.org/XML/

The XML 1.0 Standard site provides access to the text of the standard itself, details on the state of various XML working groups, and links to XML-related technologies currently being developed by the W3C.

The Network News Transport Protocol (NNTP)

www.ietf.org/rfc/rfc0977.txt?number=977

The Internet Engineering Task Force is responsible for maintaining the specification of the NNTP, which is the major underlying protocol used by Usenet. Usenet was used as an example of one of the earliest rudimentary P2P applications in Chapter 1.

Block Extensible Exchange Protocol (BEEP)

www.ietf.org/rfc/rfc3080.txt

The Block Extensible Exchange Protocol provides a way for peers to simultaneously and independently exchange messages, usually formatted as MIME or text content, using the BEEP-defined framing mechanism.