JXTA™ in a Nutshell
By Li Gong, Scott Oaks, Bernard Traversat
Preface

This handbook is a desktop quick reference for experienced Java programmers who want to develop JXTA applications. We assume that readers of this book have a good working knowledge of how to write programs in Java and some understanding of the complexities of distributed programming.

This book is based on the JXTA 1.0 specification and its implementation by Sun Microsystems with contributions from the open source community. Examples in this book are based on the Java programming language; while JXTA applications may be written in a variety of languages, JXTA programming concepts are more easily explained by using Java as the underlying programming platform. It's likely that versions of JXTA after 1.0 will contain changes that are not discussed in this book.

About the Example Programs

Most of the examples in this book deal with an online auctioning service and a user of that service. We start with a skeleton of that service and then add features to it as we proceed through subsequent chapters. The application is designed to show the features of the JXTA APIs; useful features of the application that are unrelated to the JXTA platform are left to the reader.

All examples are available online from the O'Reilly web site at http://www.oreilly.com/catalog/jxtaian/ and http://jxtainnutshell.jxta.org.

Organization of This Book

This book is divided into two sections. Part I (Chapters 1-7) is a JXTA tutorial and is designed to teach you how to write JXTA programs from scratch.

Chapter 1
This chapter introduces the basic concepts of JXTA.

Chapter 2
This chapter discusses how to download JXTA, how to set up your environment to work with JXTA, and the major concepts of working with the JXTA framework. It uses a JXTA application — the JXTA Shell — to introduce these concepts.

Chapter 3
This chapter presents the basics of a JXTA application and shows how to initialize the JXTA platform and handle simple JXTA documents.

Chapter 4
This chapter shows how the basic application can be modified to discover and create other JXTA services, including other JXTA peers and peergroups. After reading this chapter, you'll be able to write applications that can discover other peergroups, and create those peergroups if they do not already exist.

Chapter 5
This chapter shows how the basic application can be modified to use JXTA pipes for communication between two discovered peers. In this chapter, you'll learn how to discover, create, and use such pipes.

Chapter 6
This chapter shows how we can take our basic JXTA application and turn it into a service. A JXTA service can be discovered by other peers and can even be instantiated, if necessary, by a peer that needs to use the service.

Chapter 7
This chapter discusses JXTA’s security features. These include standard cryptographic procedures, such as encrypting messages and peergroup authentication mechanisms.

Part II contains quick-reference material. It includes a quick-reference guide to the Java language bindings of the JXTA APIs, a quick reference to the JXTA Shell, and the JXTA protocol specifications.
Conventions Used in This Book

By convention, we generally show Unix-style filenames and program command lines. For users of Microsoft Windows, we show the first few examples for both platforms and explain how to convert between them. When we show command lines, they are prefaced by a prompt named for the machine on which the command is running. The prompt is in constant width font, the command that you type is in constant width bold, and any output from the command is shown in constant width font. To run a command to find the version of Java used on the machine named "piccolo," we'd show a command like this:

```
piccolo% java -version
java version "1.3.1"
```

The JXTA Shell is also a command processor; its prompt looks like this:

```
JXTA>
```

Examples within the shell follow these same font conventions.

**Italic**

Used for emphasis and to signify the first use of a term. Italic is also used for commands, email addresses, URLs, FTP sites, file and directory names, and newsgroups.

**Constant width**

Used in all Java code and generally for anything that you would type literally when programming, including keywords, data types, constants, method names, variables, class names, prompts, and interface names. Constant width is also used for program output.

**Constant width italic**

Used for the names of function arguments and generally as a placeholder to indicate an item that should be replaced with an actual value in your program.

**Constant width bold**

Used for text that is typed in code examples by the user.

We’d Like to Hear From You

Please address comments and questions concerning this book to the publisher:

O’Reilly & Associates, Inc.
1005 Gravenstein Highway North
Sebastopol, CA 95472
(800) 998-9938 (in the United States or Canada)
(707) 829-0515 (international/local)
(707) 829-0104 (fax)

There is a web page for this book, which lists errata, examples, or any additional information. You can access this page at:


To comment or ask technical questions about this book, send email to:

bookquestions@oreilly.com

You can also contact the authors directly with questions by sending email to:

scott.oaks@sun.com
bernard.traversat@sun.com
li.gong@sun.com

For more information about books, conferences, Resource Centers, and the O’Reilly Network, see the O’Reilly web site at:

[http://www.oreilly.com](http://www.oreilly.com)

Acknowledgments

Many people at Sun Microsystems have contributed to the ideas and concepts described in this book. First, we need to thank Bill Joy for laying out the visionary ideas of a simpler, decentralized, XML message-based distributed computing paradigm. Mike Clary led the rest of the JXTA team that delivered the JXTA technology, and Ingrid Van Den Hoogen led the JXTA
marketing and business team that helped us refine and explain JXTA concepts. JXTA is spearheaded by a few dedicated and unconventional engineers: Mohamed Abdelaziz, Eric Pouyoul, Jean-Christophe Hugly, Mike Duigou, and Bill Yeager. We would especially like to thank them for their incredible commitment and stubborness in making JXTA a reality. This book would not exist without their hard work. We would like to also thank the entire JXTA Team: Emily Suter, Matt Reid, Juan Soto, Chris Hutcheson, Akhil Arora, David Connelly, Carl Haywood, Kuldeep Pabla, Lynn Tognoli, Laurie Wynn, Mechelle Torre, Oiching Yeung, and Cheryl Uyeda, and other Sun employees.

Many people outside of Sun have made significant contributions to JXTA. We would like to thank Chris Genly, William Bauer, Sebastien Baehni, Bruno Margerin, Daishi Kato, Dorothea Wiarda, Gerry Seidman, Jeff Altman, Nitin Borwankar, Johannes Ernst, Brendon Wilson, Scott Capdevielle, Sing Li, Kevin Burton, and finally, the entire JXTA community for having embraced JXTA with such fervor and enthusiasm.

We were very lucky to have an exceptional set of technical reviewers for this edition. The invaluable comments of Rajesh Acharya, Kevin Burton, Jean-Henry Morin, and Srikanth Raju have made this a far better book than it would have been otherwise.
Part I: Introducing JXTA

Part I introduces the JXTA platform. Its chapters contain examples to help you get started working with JXTA right away.

Chapter 1
Chapter 2
Chapter 3
Chapter 4
Chapter 5
Chapter 6
Chapter 7
Chapter 1. Introduction to JXTA

You're driving home at night; it's late and you're hungry. Imagine that your car has a device that automatically locates nearby restaurants and tells you which of them has the food you want at the best price. Better yet — what if you could contact nearby drivers, see if any of them also want to stop for food, and spontaneously form a group that negotiates with a nearby restaurant for a volume discount?

While you're waiting in the restaurant, you run into an old business acquaintance; in a brief discussion, you realize that she is working on a project that could provide a key feature for your own work. You could exchange business cards with her, of course, but why not immediately exchange the specifications of your projects instead?

And speaking of your office: you've still got to stop by the new print shop and pick up the slides for tomorrow's meeting. What if your desktop computer could find a new print shop that provides full color transparency printing services in just one hour?

These types of services have long been envisioned in an increasingly mobile and "connected" population. To a certain extent, many of them are possible today, especially if you have access to a centralized server or the other participants. JXTA will help us to expand such applications.

1.1 What Is JXTA?

JXTA is a collaborative research project for peer-to-peer (P2P) computing. "Project JXTA" (or, more simply, JXTA) stands for Project Juxtapose; it is an acknowledgment that while client/server and web-based computing are important standards today, a strong need exists for P2P computing standards for a variety of applications. These applications share a few common traits.

- They are spontaneous in nature.
- They are more effective when the supporting architecture involves many distributed providers of the service rather than a single, centralized server.
- Users of the application may be providers or consumers of the service abstracted by the application; they often perform both roles.
- Users of the application come and go and therefore may not be available at any particular point in time.
- Users of the application may be using any device at any time in any location.

Applications with these requirements are difficult to implement with traditional client/server computing models; they run better from a collection of cooperating peers with the ability to discover each other independently and to communicate regardless of the network infrastructure on which they reside.

To that end, JXTA defines a small set of common protocols that enable P2P computing. Each protocol is simple to implement and integrate into existing systems. This set of protocols defines JXTA.

It's important to realize the ramifications of that last statement. In particular, JXTA is not an API; although there are standard language bindings for the JXTA protocols, their use is not mandatory. Application developers and hardware providers may provide their own language bindings. New languages can evolve and provide their own bindings to the JXTA protocols, and new platforms can be developed that implement the JXTA protocols in entirely new ways. JXTA is language- and platform-independent.

Furthermore, JXTA is network-independent: the JXTA protocols can be transmitted over TCP/IP, HTTP, Bluetooth, home networks such as Home-PNA, and so on. Peers located on different networks can easily communicate using the standard JXTA protocols.

Hence, JXTA is a set of language- and network-independent protocols that enable the deployment of applications and services among continually changing groups of cooperating computing devices.

1.1.1 jxta.org
JXTA is an open source project; it is overseen by Sun Microsystems (which submitted much of its initial specifications and technology), but has been developed by contributors throughout the world.

All work on JXTA is organized through the JXTA web site: http://www.jxta.org/. You can download specifications, API libraries, and sample code from there; more importantly, you can sign up on this site to become a JXTA contributor. Contributors to JXTA can work on the JXTA specifications, the language bindings, or any number of collaborative JXTA applications. Also, Sun and other industry groups sponsor JXTA developer conferences.

All code developed under Project JXTA (including code originally submitted by Sun) is subject to the Sun Project JXTA Software License, which is available at http://www.jxta.org/project/www/license.html. This license is based on the Apache Software License Version 1.1 and is substantially different from other licenses under which Sun issues Java technology. The Project JXTA Software License allows all JXTA code to be redistributed in both source and binary form, provided that certain copyright and other acknowledgments appear in the redistribution.

Note that the JXTA license allows liberal use of JXTA technology but does not require that all code using JXTA adopt the JXTA license. Proprietary code can use JXTA code (provided it supplies the appropriate copyright notice) without a fee and remain proprietary.

For more information, see the licensing FAQ at http://www.jxta.org/project/www/docs/DomainFAQ.html#licmodel.

### 1.2 Why JXTA?

The rationale for JXTA has two facets. The first is the advantage of a P2P architecture for many applications. Take the auctioning example we used at the beginning of this chapter. We love the centralized auction services provided by eBay (and others) as much as anyone, but that model isn’t well-suited for the type of auction described above.

No single factor determines when a P2P architecture is appropriate. At the risk of overwhelming you with buzzwords, we can say that a P2P architecture should be considered whenever you require scalable, robust, dynamic, fault-tolerant, spontaneous, self-organizing network computing. P2P architectures achieve buzzword compliance through mechanisms other than client/server computing:

**Scalability**

A web auction for specialized French fries available only at your local restaurant presents enormous scalability problems for a centralized system, which must handle requests and responses from every conceivable restaurant and potential diner. The service can segregate requests, but this isn’t easy: the granularity the service uses to segregate the requests will directly affect the quality of the service it provides. And such a segregated service will inevitably be problematic for users on the segregation boundary. For example, if the service is segregated by zip code, users may be directed miles away within the same zip code when a closer choice is available a few blocks away in a different zip code.

A P2P system has a much easier time of it: the diner can discover nearby services automatically and at an appropriate scale. Since no centralized system monitors everything, there are no problems with system capacity: users of the service monitor nearby services in which they are interested and discard other information.

**Robustness**

As all computer users can attest, network computing is inherently unreliable: connections drop, servers crash, companies go out of business, and so on. Web search engines are useful entities and quite good at what they do, but they often lead to dead links and moved documents. We’d hate to direct our diner three miles out of his way to a restaurant that closed last week; it would be much better if we can direct him only to open restaurants. Similarly, we’d hate for our diner to go hungry because the restaurant auctioning server has gone down or is being subjected to a denial of service attack.

In a P2P architecture, it’s easier to deal with the vagaries of the network. If one particular restaurant goes offline, it can be dropped from the community. And since there’s no centralized resource controlling everything, the P2P community is less affected by
network failures or denial of service attacks. These problems will still affect individual participants in the P2P community, but the failures they cause will not have a widespread effect on others within the community.

**Dynamic behavior**

P2P applications discover services and other resources dynamically and automatically when they are needed. There is no central registry of these services. This dynamic behavior allows scalability and robustness across the entire system, of course, and it also influences the way in which you must think about P2P applications. Resources may be available when you need them, but they may not; like the print shop posited above, they may be available just in time.

This adds a certain complexity to a good P2P application and would seem to place a big burden on the developer, who must explicitly cope with services that are not always available. In the end, however, this leads to better programs: a network is a changing entity, and applications that embrace and account for those changes are better suited to run in that environment.

**Fault resistance**

Well-written P2P applications that take advantage of the dynamic nature of the network have a degree of fault resistance that most applications do not: the P2P application is tolerant of faults in other services and in the network. Fault tolerance often implies a level of hardware redundancy; that definition does not necessarily apply to JXTA (though there is no reason, of course, why a JXTA service could not run on fault-tolerant hardware).

**Spontaneity**

All of our examples at the beginning of this chapter had a certain degree of spontaneity to them. This is a natural outgrowth of the dynamic behavior of a P2P community; it gives an important advantage to a P2P architecture, particularly when participants in the community are mobile.

**Self-organization**

Because they do not rely on a central registry, P2P communities are self-organizing. As our diner contacts others, they can self-organize into a group that can leverage their buying power when approaching a restaurant. In a general sense, a P2P community is organized around a group of peers who are interested in the same service(s).

As we develop sample JXTA applications later in this book, we’ll expand on these features of a P2P architecture.

The second facet of the JXTA rationale stems from what JXTA gives us: a common platform on which P2P applications can be built. There are several popular P2P applications in use today: various instant messaging platforms, file-sharing systems such as Gnutella, and so on. Traditionally, these applications have been developed from the ground up and are incompatible with each other; users in one P2P community are isolated from users in a different community. JXTA seeks to provide a common platform on which P2P applications may be built.

The use of a common platform allows P2P applications to share information easily. In addition, the JXTA platform allows developers to create P2P applications easily, since the JXTA protocols provide an easy means for dynamic discovery, message passing, and other core features of P2P architecture. In fact, JXTA provides a true P2P platform on which applications can be built: most of the applications that are considered P2P applications do not have all of the advantages we listed above. Napster and AOL Instant Messaging, two of the most popular P2P applications, rely on centralized registries; they suffer somewhat in the areas of robustness and fault tolerance, where true P2P applications can excel. JXTA allows you to develop true P2P applications.

### 1.3 JXTA Overview

At a specification level, JXTA defines a number of protocols that enable P2P computing. At another level, the JXTA community has developed language bindings; these bindings implement the protocols and allow developers to write JXTA applications. Embedded within these bindings are several core JXTA services that provide needed functions for JXTA applications. Neither the language bindings nor the core service implementations provided by the JXTA community are strictly necessary in order to develop JXTA applications: as long as the application correctly implements the necessary protocols, it can use any language, network transport, and service...
implementation to do so. Nonetheless, we expect most initial JXTA applications to follow the Java language bindings presented throughout this book and to use the core services implementations provided by the JXTA community.

Think in terms of other protocol specifications. Today, most computer networks run on the Internet Protocol (IP) specification. Other network protocols, like the Transmission Control Protocol (TCP) and Uniform Datagram Protocol (UDP), are layered on top of IP. Still other protocols, such as the HyperText Transfer Protocol (HTTP), are layered on top of those protocols. Yet for the most part, developers of network applications and services have no idea how the underlying protocols work. Developers create a TCP connection to another server by opening a socket with the BSD socket library, the WinSock socket library, the Java Socket class, or some other API layer. Some intrepid developers may use TCP sockets to send HTTP protocol messages, but most will use an HTTP protocol binding, such as the Java URL-related classes. Still fewer developers will open raw IP sockets and deal with the network at that level.

JXTA is a network protocol that is layered on top of other network protocols. In its Java language bindings, it is layered on both TCP and HTTP, but it can be layered equally as well over emerging network standards, such as Bluetooth. You could certainly open TCP sockets and send JXTA protocol messages over those sockets, but it is far easier to use the JXTA language bindings to hide the details of the protocol.

There are similar layers within the language bindings as well. The primary mechanism that JXTA peers use to communicate with each other is the JXTA pipe, and there are protocols specifying how JXTA pipes work. The protocols depend on JXTA endpoints, which are essentially network connections (for which there are other JXTA protocols). However, the JXTA pipe classes shield the developer from needing to know about the JXTA pipe protocols, the JXTA endpoint protocols, and the JXTA endpoint classes.

Our intent in this book is to give an overview of the underlying protocols and features of JXTA and to focus on the high-level Java language API bindings that use those protocols. These are the bindings that you'll need to implement most types of JXTA applications and services. If you're interested in moving JXTA to other networks or developing other language bindings, you'll need to consult the JXTA specifications that are online at spec.jxta.org and are reprinted in the reference section of this book.

In Chapter 2, we use a standard JXTA demo application—the JXTA Shell—to step through and understand the key features of the JXTA platform. These include:

**Peers**

Peers are the basic unit of JXTA. They are any type of networked device, although in JXTA we often make the distinction between a peer that provides a service and one that uses a service. The former is referred to as a service while the latter is referred to as an application. However, all services are applications: a peer that provides a service from which you can order French fries is itself a consumer of a service that provides peer discovery. Generally, peers can be both producers and consumers of services.

**Peergroups**

Peers self-organize into peergroups. A peer must belong to a peergroup before it can communicate with other peers (and it can communicate only with peers that have joined the same peergroup). A peer can join multiple peergroups.

**Pipes**

Pipes are the basic tool that JXTA peers use to communicate with each other. They are somewhat analogous to UDP sockets: they send or receive chunked pieces of data.

**Advertisements**

Advertisements are special XML documents that announce the presence of JXTA resources. A JXTA resource can be just about anything used by a peer: other JXTA peers, peergroups, pipes, and services (such as an auctioning service) provided by JXTA peers.

**Discovery**

Discovery is the basic process by which peers locate advertisements and, consequently, locate resources that they can use.

1.4 Summary
P2P applications have caught the imagination of millions of computer users worldwide; from Napster to Instant Messenger, P2P applications are among the most widely used applications on the Internet today.

Yet there are several barriers to the widespread adoption of P2P technology. Primary among these is that such applications tend to be developed in isolation and without standards; an application that can deal with IM has no concept of how to operate in any other P2P realm. In addition, though much of the communication in these applications is P2P, they still tend to rely on big, centralized servers to provide key features of the service.

JXTA seeks to level these barriers by providing a standard set of protocols that all P2P applications can use. These protocols are network- and language-independent and allow the development of services that do not rely on a centralized system. In the same way that standardized network protocols such as TCP and HTTP made the explosion of client/server network applications possible, the standardized JXTA protocols seek to fulfill the promise of P2P computing.

In the next chapter, we’ll explore JXTA in more depth by interacting with a standard JXTA demo, the JXTA Shell. This will position us to begin writing our own JXTA applications in the following chapters.
Chapter 2. Getting Started with JXTA

In this chapter, we’ll see how to get started with JXTA. Although JXTA is a language- and platform-neutral specification, we’ll focus on using the standard JXTA applications for the Java platform. The basic concepts that you’ll learn in this chapter are applicable to any JXTA implementation using any language; we chose to illustrate the concepts of JXTA using the Java platform because it allows for the simplest discussion of JXTA concepts, and because the Java platform gives us a common basis for our examples, regardless of the computer on which you might run them.

We’ll start by discussing how to set up a Java environment to run JXTA programs. Then we’ll look in depth at one particular program: the JXTA Shell. Examining the shell will allow us to look in depth at each of the protocols and techniques that JXTA defines; working through the examples in this chapter should provide you with a working knowledge of the key concepts of the JXTA platform and how programs operate within that platform.

2.1 Setting Up a Java Environment

The first step in using JXTA is to set up your environment. In this case, that means setting up a Java environment to run JXTA, for which you’ll need three things: a Java platform, the JXTA Java class libraries, and any JXTA programs that you want to run.

2.1.1 The Java Platform

For the Java platform, you’ll need the Java 2 Standard Edition (J2SE), Version 1.3.1 or later (Version 1.4 is preferred). Multiple implementations of the JXTA protocols are already available. A C implementation (jxta.c.jxta) and a Personal Java Profile edition are both available. This chapter focuses on running and explaining existing JXTA applications. Therefore, if you’re using the Java 2 platform, you need only the Java 2 runtime environment (J2RE). If you plan on programming with JXTA (using the examples in subsequent chapters), then you’ll need the Java 2 Software Developer’s Kit (SDK).

There are a number of ways to obtain current releases of the Java platform. If your system is running Solaris, Microsoft Windows, or Linux, the simplest way is to download the SDK from http://java.sun.com/j2se/. For other operating systems, check with your system vendor. More commonly, many integrated development environments (IDEs) come with support for Java (and hence a Java platform).

Once you’ve obtained and installed Java, you must make sure that the java executable is in your standard path.

2.1.2 The JXTA Class Files and Programs

You can obtain all the JXTA files you need at http://download.jxta.org/easyinstall/install.html. On this page, you can obtain the JXTA demo package for a variety of platforms. In fact, the JXTA demo implementation at this site is written completely in Java; the difference between the platforms lies only in how the parts of the implementation are packaged and how they are installed. Therefore, for Microsoft Windows, download an executable (.exe) file; for Solaris, download a shell script, and so on.

When you execute the installation program, you are prompted for a directory in which to install the code. On Unix systems, the default directory is /JXTA_Demo; on Microsoft Windows, the default directory is C:\Program Files\JXTA_Demo. Within the directory you select, the installation creates the following:

```
lib
```

This directory contains a set of jar files that contains the JXTA implementation and another set that contains implementations of the JXTA demo applications.

Shell
The JXTA Shell is an interactive application that lets you look at the JXTA environment and try out basic JXTA functionality. We'll examine the shell in detail later in this chapter.

**InstantP2P**

This is another sample JXTA application; it contains functionality to chat one-on-one, chat with a group, and share files. This application uses all of the standard facilities of JXTA, so it is a good example on which to model other JXTA P2P applications.

This is all you need to use JXTA technology, both as an end user and as a developer. If you're going to do JXTA development, you should add each of the jar files in the *lib* directory to your classpath. If you're simply going to run the sample applications, there are scripts in each application directory that set up the classpath and run the application.

In our examples throughout this book, we assume that you've installed this hierarchy into */files/JXTA_Demo* (*C:\files\JXTA_Demo*). We'll also assume that your classpath contains the current directory and the necessary jar files from the *lib* directory:

- */files/JXTA_Demo/lib/jxta.jar*
- */files/JXTA_Demo/lib/beepcore.jar*
- */files/JXTA_Demo/lib/cryptix-asn1.jar*
- */files/JXTA_Demo/lib/cryptix32.jar*
- */files/JXTA_Demo/lib/jxtaptls.jar*
- */files/JXTA_Demo/lib/jxtasecurity.jar*
- */files/JXTA_Demo/lib/log4j.jar*
- */files/JXTA_Demo/lib/minimalBC.jar*

As you become more familiar with JXTA, you may want to get involved with other JXTA projects, use other JXTA applications, or examine the JXTA source code. You can download all of these things from [http://www.jxta.org/project/www/download.html](http://www.jxta.org/project/www/download.html).

### 2.2 Basic JXTA Concepts

Now that we have all of this software, we'll use it to explain a little more about the basic JXTA concepts we outlined in Chapter 1, including how a JXTA application is constructed. We'll use the JXTA Shell as the basis for our exploration, since it provides us with an interactive tool that uses the JXTA platform to perform its operations.

#### 2.2.1 JXTA Shell Syntax

Before we dive into the shell, here are some notes on its syntax. Like any shell, the JXTA Shell issues a prompt (*JXTA*>) at which you type in commands.

Shell commands have two kinds of output. Most of them simply send their output to the screen. Some commands, however, produce an object as their output. These objects should be saved in a shell variable. If you do not save the object, most commands will create a new object with a default name to hold the return value; if you're going to need the object, it's easier to assign a name to it yourself. Shell variables are created in JXTA by assigning a new name to the output of such a command.

Here are some simple examples. The `env` command produces as its output a list of all the shell variables and their values:

```shell
JXTA> env
stdin = Default InputPipe (class net.jxta.impl.shell.ShellInputPipe)
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 = Console OutputPipe (class net.jxta.impl.shell.ShellOutputPipe)
consin = Default Console InputPipe (class
```
Shell variables are created by assigning a new name to the output of a command that creates an object. `mkadv` is such a command; here we store the object it creates in the `myadv` shell variable:

```
JXTA> myadv = mkadv -p
```

You can print out the content of certain variables by using the `cat` command. If the variable has structured data, `cat` will print it out:

```
JXTA> cat myadv
<?xml version="1.0"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement>
  <id>
    jxta://59616261646162614A757874614D50474168B1395E034DEA90F3BC8CD7D
    361840000000000000000000000000000000000000000000000000000000000000401
  </id>
</jxta:PipeAdvertisement>
```

A list of all shell commands can be obtained via the `man` command; the `man` command can also print out help for a specific command (e.g., `man mkadv`). A complete shell reference appears in Chapter 12.

### 2.3 JXTA Peers

We'll start our examination of JXTA's key concepts with the notion of a peer. As defined in the JXTA specification, a peer is a device that implements one or more JXTA protocols. The key idea is that something implements the protocols. Don't get too hung up on the notion of a device—a device is not necessarily a machine. A single machine can host multiple JXTA programs, each of which is really a peer. The program is really a virtual device.

Of course, peers normally communicate with each other over a network, so a JXTA community typically has many different machines within it. However, some of these machines could be running multiple JXTA peers. A single JXTA peer could be a distributed application running across multiple machines. And if you're following the example in this chapter at home on your own single computer, you'll end up running many peers on your local system.

To start our first peer, we'll execute the JXTA Shell. On Microsoft Windows systems, you can do this by selecting Programs → Jxta → JXTA Shell from the start menu. On other platforms, you can do this by executing these commands:

```
piccolo% cd /files/JXTA_Demo/Shell
piccolo% sh shell.sh
```

When you first start the shell, you'll see the window displayed in Figure 2-1. This allows you to configure the shell for the network on which it will run. We explain the configurator in detail later in this chapter. For now, we'll just step through the two required elements that you must configure.

---

[1] This default configuration assumes that your computer is directly connected to the Internet. If you are not connected to the Internet directly or connect to the Internet through a proxy, then you must take additional steps as outlined in the detailed section on the configurator tool later in this chapter.

---

**Figure 2-1. The JXTA configurator tool**
First, fill in the peer name with a string of your choice (we chose "Test Shell 1") and press the OK button. On the next screen, fill in the username you'd like to use and a password for that username. In Figure 2-2, we've specified myusername and entered a password.

Figure 2-2. Entering a username into the configurator

The next window that comes up is the JXTA Shell itself. This is the first JXTA peer that we've created. The shell is interactive; its command prompt is JXTA>, and it accepts more than 40 commands. For now, we'll just look at the commands that illustrate the basic JXTA concepts. One of these commands is the peers command; it displays a list of all the peers that the shell knows about. If you execute that command now, you'll see the following:

JXTA> peers
peer0: name = Test Shell 1
peer1: name = JXTA.ORG 237
peer2: name = JXTA.ORG 235

More information can be printed this way:

JXTA> peers -1
peer0: ID = uuid-
59616261646162614A78746159325933A69CA85997D6433D82BDA4A4D0953AC603 name =
Test Shell 1 addr = tcp://192.168.1.101:9701/
peer1: ID = uuid-
59616261646162614A787461593259336ACC981CF047CFA name =
JXTA.ORG 237 addr = tcp://209.25.154.237:9701/
peer2: ID = uuid-
59616261646162614A7874FAEDBBB84753E1E16503C14495AE name =
JXTA.ORG 235 addr = tcp://209.25.154.235:9701/

All peers have names (in the case of our peer, "Test Shell 1"). In addition, peers carry other important pieces of information: the peergroup to which they belong (which we'll explain later in this chapter), a unique peer ID, and information about what network addresses can be used to contact the peer. You can see this information in the shell with the whoami command:
The peer ID is unique across all JXTA peers. The transport addresses contain the information about how the peer may be contacted.

You can discover a peer explicitly using its peer ID like this:

```
JXTA> peers -p jxta://5961626164616261A78746159325933A69CA85997D6433D82BDA4A4D0953A03
```

This technique is easier to imagine programatically, where you wouldn’t have to type in the entire string. You can, for example, save a peer ID and use it in later sessions as the default peer to provide a service (assuming that the program can handle the fact that the default peer may no longer be available); in fact, peer IDs are often saved by the JXTA platform itself when it performs caching.

Peers keep configuration information within their current directory. If you want to start a second instance of the shell on the same machine, you cannot do so from the same directory. Instead, you must follow the instructions on advanced configuration that we give later in this chapter.

### 2.4 Peergroups

Peers self-organize into groups. JXTA does not define what these groups are, nor why they exist: the JXTA framework supports the creation of groups and the definition of group membership, but it is up to cooperating peers to define groups, join groups, and leave groups. Among other purposes, peergroups serve to:

**Define a set of services and resources**

In order to participate in an auction, a peer must find a JXTA peergroup that provides an auction service. The peer must join that group; only peers that have joined the group can use the services available in the group.

There are a number of important resources that exist within each peergroup. A service is a specific type of JXTA resource. A peer that wants to use the resources of a particular peergroup must join that peergroup. Similarly, peers can communicate only with each other if they have joined the same group (although, as we’ll see, all peers belong to the NetPeerGroup, so they can always communicate with each other).

**Provide a secure region**

Because of their membership requirement, peergroups form an entity with a logical boundary. Peergroups may strongly enforce a membership requirement so that the boundaries define a secure environment as well: content within a peergroup can be accessed only by peers belonging to that group, so a peergroup with strict membership requirements can secure its content from the rest of the world.

The underlying network topologies of peers within a peergroup do not necessarily reflect physical network boundaries, such as those imposed by routers and firewalls. A peergroup can consist of peers on completely disparate networks. Peergroups virtualize
the notion of routers and firewalls, subdividing the network in secure regions without respect to actual physical network boundaries.

**Create scoping**

Peergroups are typically formed and self-organized based on the mutual interest of peers. No particular rules are imposed on how peergroups are formed. Peers with the same interests tend to join the same peergroups. Therefore, the abstract regions defined by a peergroup provide an implicit scoping mechanism for peergroup operations. Peergroup boundaries delimit the search horizon when looking for peergroup resources. Peergroup scopes help scalability by reducing network traffic explosion within the JXTA network. Messages within a peergroup are propagated only to peers that are members of the peergroup.

**Create a monitoring environment**

Peergroups enable peers to monitor members, interactions (traffic introspection, accountability, traceability, etc.).

The JXTA platform defines two kinds of peergroups:

**The NetPeerGroup**

This is the default peergroup a peer joins after booting the JXTA network; it is sometimes called the World Peergroup. Configuration of the NetPeerGroup is typically done by the administrator in charge of the network domain on which the peer is located. The NetPeerGroup defines the initial scope of operations of a peer and the default services it provides.

**User peergroups**

Users (and applications) can create new peergroups with their own set of customized services and membership policies. User peergroup services are created by either cloning and extending the NetPeerGroup services or by creating new ones. All user peergroups are subsets of the NetPeerGroup.

All peers are automatically members of the NetPeerGroup peergroup; they do not need to take any special actions to join the group, and they cannot leave the group. Peers may opt to join and leave other groups at will. In order to join a group, a peer must discover the group and then ask to join it. Depending on the group, the peer may have to present authentication credentials, and the group will vote on the peer's application.

Let's see how this all works in the context of the shell. When the shell starts, it automatically joins the NetPeerGroup. One way that you can find the group that a shell belongs to is with the `whoami -g` command:

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

Any peer can create a peergroup. Peergroups are typically composed of peers that have either a common set of interests (e.g., music or baseball peergroups) or that interact with each other to exchange contents.

### 2.4.1 Peergroup Services

Peergroups allow us to introduce the notion of protocols and services in the JXTA platform. A peergroup provides a set of services that peers can utilize. The JXTA platform defines seven core services:

- Discovery service
- Membership service
- Access service
- Pipe service
- Resolver service
- Monitoring service (sometimes called the "purines" service)
- Rendezvous service
This set of core services is used to "seed" the peer so it can discover and find enough resources to support itself. The NetPeerGroup services provide sufficient functionality for allowing this dynamic bootstrapping mechanism. The core services were selected to enable a peer to have enough functionality to create and join a large variety of peergroups:

- The membership service is necessary to enforce a policy for new peers to join a peergroup.
- The discovery service is necessary for a peer to discover new network resources (so a peer can expand its capabilities).
- The resolver service provides the essential infrastructure for services to communicate.
- The pipe service provides the ability for one or more peers to exchange messages.

User peergroups can extend, replace, or add new services, allowing a variety of peergroups to be created. A peergroup does not need to provide all of these services; it can provide only those that are necessary to support the peers within the group. Most peergroups, however, will provide at least a membership service. Since peers are always members of the NetPeerGroup, they can always obtain these services from the NetPeerGroup. The NetPeerGroup provides all seven of these services. In general, each of these services defines a specific protocol used to access the service.

2.4.2 Creating Peergroups

Any peer can create a peergroup. The shell creates a peergroup with these commands:

```
JXTA> mygroupadv = mkadv -g mygroup
JXTA> mkpgrp -d mygroupadv
```

The first of these commands creates an advertisement (mygroupadv) of a new group with a symbolic name of mygroup (advertisements are discussed in more detail later in this chapter). The mkpgrp command actually creates the group.

When you create a group in this manner, information about the group is broadcast to other peers using a discovery protocol that we discuss in the next section. In general, a peer can find a peergroup in two ways:

- When a new group is created, an announcement of that group is published, and peers who are searching will discover the new group.
- A peer can initiate discovery to find groups. In the shell, this is accomplished with the groups -r command:

```
JXTA> groups -r
```

You can see which groups the shell knows about by using the groups command:

```
JXTA> groups
group0: mygroup
group1: anothergroup
```

In this last example, we know about mygroup because that's the group that we created; we know about anothergroup because another peer on the network created that group and we discovered it. Note that the NetPeerGroup does not show up in this list, even though it is also a peergroup.

Each peergroup is uniquely identified by a unique peergroup ID. The peergroup ID is guaranteed to be unique within the entire JXTA network.

2.4.3 Joining Groups

Once a peer has discovered or created a group, it can join the group. This entails sending an application (with any necessary credentials) to the group and awaiting a response. The
membership service of the group will vote on the new application and send back either an acceptance or a rejection.

A shell joins a particular group by using the `join` command:

```
JXTA> join -d group0
```

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

1Identity: jra

After executing this command, the shell is a member of the group mygroup. A quick note about syntax here: `group0` comes from the list of groups we obtained with the `groups` command. You can use the symbolic group name (e.g., mygroup) to join a group only if that shell created the group. To join a group that the shell discovered (such as anothergroup), however, you must use the `group#` that the shell has assigned. Because the shell caches information about groups, it might have created the group, exited, and restarted. In that case, you must also use the `group#` that the shell has assigned.

A peer can join multiple groups by executing another `join` command; the `join` command can also be used to determine which groups the peer has joined:

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

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

1Identity: sdo

JXTA> join

Joined group    : mygroup
Joined group    : anothergroup    (current)

Commands that the shell executes will be associated with the current group (anothergroup); the current group can be changed with the `chpgrp` command:

```
JXTA> chpgrp mygroup
```

In this command, you always use the symbolic group name rather than the `group#` name. The current group is used to find any service that the shell needs; it is also the group printed by the `whoami` command.

Membership in any group created by the shell is automatic, so these examples work well when dealing with groups created by two different shells. A shell might also discover a different group, however: one that has membership criteria and requires the shell to present some special credentials. Membership in such groups requires that you know how to fill out a credential request, which is group-specific; we'll look at how this works in Chapter 7.

### 2.4.4 Canceling Group Membership

A peer can cancel its membership in a group. The shell cancels its membership in a group with the `leave` command:

```
JXTA> leave
```

This command cancels membership in the current group. You cannot leave the NetPeerGroup.

### 2.5 Discovery

So far, our shell does not know a lot of peers. In the JXTA world, this is not a good thing, as a peerless JXTA application is not very useful. The first thing it must do, therefore, is find other peers with whom it can exchange information. The process of finding other peers is called discovery.

The process of discovery applies to any JXTA resource: peers discover other peers, and they also discover peergroups, pipes, advertisements, and other resources. There are two ways in which peers discover resources: peers discover resources dynamically, and they use special, statically configured peers to tell them about JXTA resources. Discovery occurs within the context of a peergroup: a peer attempts to discover resources within a specific peergroup.
2.5.1 Dynamic Discovery

Peers use the JXTA Peer Discovery Protocol (PDP) to discover JXTA resources dynamically. The PDP defines the lowest-level technique that is available for peers to discover resources. On an IP network, it consists of two parts: a multicast message that is sent on the local network and the use of rendezvous peers to discover peers beyond the scope of the local network. Other network bindings will behave differently, but the essential operation of the PDP is that the requesting peer sends a message, and resources that hear the request will respond directly to the peer. Therefore, a peer using PDP will discover all JXTA resources that are on the local network or are known by the peer's rendezvous peers.

The implementation of discovery is actually a feature of the peergroup in which the peer is running. Certain peergroups may define other protocols by which peers can discover resources within the peergroup. The PDP is implemented by the discovery service that runs within the NetPeerGroup (and most other peergroups).

The peers -r command of the JXTA Shell implements the PDP; it will discover all other JXTA peers on the local network. Before executing this command, you may want to start a shell on another machine on your network (if you must start a second shell on the same machine as the first, skip ahead to Section 2.6 to see how to do this).

Now in the original shell window you can execute commands to discover the second peer:

```
JXTA> peers
peer0: name = Test Shell 1
```
```
JXTA> peers -r
peer discovery message sent
```
```
JXTA> peers
peer0: name = Test Shell 1
peer1: name = Test Shell 2
```

If you execute the same commands from the second shell, you'll notice that it numbers the peers differently. The peer numbers are used only internally by the shell application; peers are not numbered in a JXTA sense (except for their unique IDs).

It can take some time for the shell to discover other peers; executing the peers -r command may not necessarily report all of the peers on the network until the shell has completed its discovery process. In addition, some peers may not respond because they miss the multicast message; no reliable network protocol is used to discover peers. On the other hand, as more peers are added to the network, discovery is faster: that's because when a peer responds to a discovery message, it sends information about itself and about all other peers that it has already discovered.

2.5.2 Static Discovery Through Rendezvous Peers

There are special JXTA peers known as rendezvous peers. These peers are like landmarks: they keep a list of peers (and other JXTA resources) that they know about, and other peers query them for that list.

If you've been following along with our example, and you're on a computer connected to the Internet, when you first executed the peers command, you received a list of peers. This is because the shell is configured to contact certain known rendezvous peers on the Internet; by default, it will automatically contact the hosts 209.25.154.235 and 209.25.154.237. These hosts showed up in our earlier example as peers JXTA.ORG235 and JXTA.ORG237.

These hosts are the bootstrapping set of rendezvous peers for the NetPeerGroup. Peers may discover rendezvous peers dynamically and use them in place of these; the intent of the bootstrapping rendezvous peers is to initiate the discovery process of dynamic rendezvous peers.

When the shell contacts the peer at one of these hosts, it will get back a list of all other peers that have discovered that same rendezvous peer. Therefore, a JXTA Shell running on a machine connected to the Internet may automatically discover many peers.

You can use the rdvstatus command within the shell to see if the shell has successfully contacted a rendezvous peer:

```
JXTA> rdvstatus
```
Rendezvous Connection Status:  
----------------------------

Is Rendezvous: [false]

Rendezvous Connections:

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

Rendezvous Disconnections:

  [None]

In this case, the shell itself is not acting as a rendezvous peer; the shell has made contact with two rendezvous peers.

There can be as many rendezvous peers as peers in the network, although a certain ratio between rendezvous peers and other peers is expected to be maintained, depending on the peergroup discovery policy. Some peergroups will have a few rendezvous peers; others will have many. Smaller peergroups typically need fewer rendezvous peers. Larger peergroups tend to need more rendezvous peers to reduce the chance of creating islands of peers that do not communicate with others.

A peer can dynamically advertise itself or remove itself from the list of rendezvous peers. Peers can discover new rendezvous peers dynamically and add them to their list of known rendezvous peers. As with the shell, any peer can be preconfigured to know about a fixed number of rendezvous peers. Each peergroup can define its own ratio of peers to rendezvous peers to support the peergroup-specific requirements.

Because a peer knows the address of a rendezvous peer, it can use whatever network protocol it wants to contact it (assuming, of course, that the rendezvous peer also supports that protocol). In the shell, the preconfigured rendezvous peers are all defined as using HTTP as their transport mechanism.

### 2.5.3 Other Discovery Mechanisms

JXTA services and applications are free to use any other technique to find each other. The shell, for example, can be configured to pass its requests to the rendezvous peers through a proxy; this is useful if you must pass through a firewall to get to the Internet. In this case, the shell wraps the standard PDP message into an HTTP request that it sends to the proxy server; the proxy forwards it just as it would any other HTTP message. When the rendezvous peer receives the request, it puts its answer back into the HTTP response and sends it to the proxy, which returns it to the shell.

Any variation on this technique is valid, as long as all the peers agree on the technique. At a minimum, peers will always be able to discover each other using the basic PDP (assuming, of course, that their network topology supports the discovery we described above).

Consider the peers shown in Figure 2-3. When the JXTA Shell joins its local network, it will send out a broadcast version of the PDP. It will get back two responses. One response will come from the JXTA searching service, and the shell will now know about the searching service. The other response will come from the JXTA rendezvous peer. Because the rendezvous peer is sitting on a machine that is attached to both the local and corporate networks, it has previously discovered the searching peer and the file-sharing peer. When the rendezvous peer receives the PDP broadcast method from the shell, it will tell the shell about these two peers.

The shell will then send a PDP message via HTTP directly to JXTA.ORG237. The rendezvous peer at JXTA.ORG237 will then tell the shell about all the peers that JXTA.ORG237 has discovered; this is how the shell can discover the JXTA auction service. In Figure 2-3, the HTTP message passes through a proxy in order to pass through the firewall.

**Figure 2-3. JXTA peers and discovery**
Passing through the proxy is a value-added feature of how the JXTA Shell is programmed (and a value-added feature of the standard Java language bindings that we use throughout this book). At a basic level, the PDP works over HTTP; if we were to remove the firewall from this network, then the JXTA file-sharing service could use PDP directly to contact JXTA.ORG and discover the JXTA auction service.

Also note how peer discovery is improved as more peers are added to this example. When a new peer is connected to the local area network and executes the basic PDP to find peers dynamically, it will immediately find the JXTA auction and file-sharing services. That's because the shell has already found these services. When the new peer sends its discovery message, the shell will respond with all the peers it knows about.

2.5.4 Relay Peers

JXTA peers can elect to become relay peers. Relay peers send dynamic discovery requests across different networks.

In Figure 2-3, the JXTA rendezvous peer sits on two different networks. The JXTA Shell discovered the JXTA file-sharing service because it knew about the rendezvous peer and contacted it directly. If the JXTA rendezvous peer was a JXTA relay peer instead, then when the JXTA Shell performed its multicast discovery, the JXTA relay peer would have seen the multicast message and routed it onto the corporate network. This would have been seen by the JXTA file-sharing service, which would have sent a message back to the shell.

The end result here is the same: the shell discovers the service, but the procedure is quite different. In the case of the rendezvous peer, the shell had to have explicit knowledge of the rendezvous peer in order to contact it. In the case of the relay peer, the discovery would happen dynamically.

In practice, peers often serve as both relay and rendezvous peers.

2.6 JXTA Application Configuration

The dialog box that we showed in Figure 2-1 should make a little more sense now. That box is called the JXTA configurator tool, and it’s a basic feature of the JXTA platform. All JXTA applications use this tool when they are first configured.

Configuration information is stored by this tool in the local directory. The configuration information also includes a local port number on which the JXTA application will listen. For that reason, you cannot run two instances of the shell (or any other application) on the same machine from the same directory. Instead, you must do the following:

- Create a new directory in which to hold the configuration information (e.g., /files/JXTA_Demo/Shell.2).
- Copy the shell.sh or shell.bat script into that directory.
- cd to that directory.
- Execute the startup script.

When this shell starts, you will again be presented with the configuration box from Figure 2-1. You’ll need to perform some advanced configuration in order for this second shell to run. The configurator has four panels:

**Basic**

The basic panel of the configurator allows you to assign a name to your peer. Any string can be entered (we chose “Test Shell 1”): it can represent a person's name, a hostname, or any other identifier. The name you enter is not guaranteed to be unique. No centralized naming service is used by the configurator; therefore, two peers may end up with the same name. This is okay, since the peer name is intended to be used only as a hint for searching or identifying peers. Also, each peer is assigned a peer ID that is guaranteed to be unique for every peer in the JXTA network.

This panel also allows you to specify whether your machine is behind a firewall. If you are behind a firewall, select the option to use a proxy server and enter the proxy server's name and port into the given boxes. When a proxy server is enabled, requests to the rendezvous peers will be wrapped into HTTP messages before they are sent to the peer.

**Advanced**

The advanced panel allows you to set up network information about the peer. JXTA v1.0 supports two transports: TCP/IP and HTTP. By default, the configurator tool preconfigures both a TCP/IP and HTTP transport to communicate with other peers. You should enable the TCP settings if there are other JXTA peers on your local network that you want to discover through the broadcast mechanism of PDP. The choice menu lists all the networks that the machine knows about; these are the networks on which the shell could listen for broadcast messages from other peers. Note, however, that you can select only one network on which to listen for messages.

The text box in the TCP/IP configuration section contains the network port that will be used for application-level messages to other peers. If you are running multiple instances of the JXTA platform on a single computer, you must change the port number parameter: each JXTA application needs to have its own port number to bind a listening socket to the port.

Similarly, enable the HTTP settings if you want to make connections to the rendezvous hosts we listed above. Note that you enable this to use HTTP to rendezvous to those hosts regardless of whether you're behind a firewall or use Network Address Translation (NAT). The message here is a little misleading. If you're behind a firewall, then you should enable the proxy server on the basic panel and fill in its name and port number as appropriate. Making connections through the HTTP rendezvous ports can take some time; while you're experimenting with JXTA on your local network, it's more efficient to disable this feature (though you'll lose the ability to interact with other peers on the Internet while this feature is disabled). If you are not connected to the Internet, you must disable the HTTP settings.

One more note about the TCP port. Peers on a local TCP network discover each other through a multicast message. However, once they've discovered each other, they communicate directly over a specific port (known as the endpoint). The value of the endpoint is embedded within the PDP messages that the peers exchange. A peer listens for connections on this port; one peer contacts another by making a connection to the second peer's endpoint. That's why we had to change the port number when we started a second shell on the same machine: each shell needs an unused port number on the machine. If you run two shells on different machines, changing the port isn't necessary.

**Rendezvous/relays**

The rendezvous/relays panel allows you to specify specific hosts to be used as rendezvous and/or relay peers. By default, JXTA peers will download the lists of HTTP rendezvous peers by connecting to http://rdv.jxtahosts.net/cgi-bin/httpRdvsProd.cgi and HTTP relay peers by connecting to http://rdv.jxtahosts.net/cgi-bin/routersProd.cgi. If you press the Download relay and rendezvous lists button at the bottom of this panel, you can specify a different location from which to load these lists. When the lists are loaded, you'll see that
the default addresses for JXTA.ORG 235 and 237 have been added as both rendezvous and relay peers. You may delete either of these peers or add any other peers through the fields on this panel.

Note that if you disabled the HTTP transport on the advanced panel that the HTTP rendezvous and relay settings are not available.

Security
The final panel is one we’ve seen: it requires you to enter a name and password for access to this peer. Whenever the peer is restarted from this directory, you will be prompted for the values you entered into this panel.

2.6.1 Re-Entering Configuration Information

Network configuration information that you enter in the configurator is stored in the PlatformConfig file of the current directory. When a JXTA application starts, it consults this file to determine how it should configure itself. Information about your username and password is stored in files within the ./pse directory.

The first time that you start an application in a directory, the configurator will run and automatically create those files. After that, if you need to change the configuration information, you must remove the PlatformConfig file (to change the network information) and/or the pse directory (to change the username and password). Alternately, you can create a file called reconf in the current directory, in which case the configurator will run, and you’ll be allowed to change the network information (but not the username and password).

From the shell, you can create the reconf file with the peerconfig command:

```
JXTA> peerconfig
```

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

Then exit the shell and restart it; the configurator will reappear, and you can enter new network information.

2.6.2 Firewall and NAT Traversal

A fundamental problem P2P systems such as JXTA need to address is the ability to create a connection between two peers located at the edges of the Internet. Firewalls and NAT solutions currently deployed on the Internet create logical partitioned zones that preclude peers from talking directly with each other.

Firewall solutions are primarily deployed to create protected regions and to limit incoming traffic from the outside world. Many companies on the Internet today use firewall software to protect their internal networks from unwanted external intruders and traffic; home firewalls are becoming increasingly popular for individual users as well.

NAT solutions address the limitation on the number of IP addresses that can be assigned on a particular local network. These limitations stem from two areas. First, the sheer number of devices that are connected to the Internet is greater than the present address space of IP addresses. Second, many users (particularly home users with small networks) prefer to use NAT to shield information about their network topology from the outside world and to limit the number of IP addresses they must purchase from their service provider. IPv6 will eventually solve the first of these issues, but not the second.

So both firewalls and NAT address real problems; they will be a reality of public networks for the foreseeable future. P2P networks such as JXTA need to provide traversal and piercing capabilities to enable any peer on the network to talk with any other peer. JXTA assumes uniform reachability between peers, independently of their physical location in the network (e.g., behind a firewall or NAT).

Therefore, JXTA implementations must support firewall traversal and NAT piercing. This is accomplished in JXTA 1.0 using HTTP; it’s important to set the fields of the configurator correctly to take advantage of these features.

2.6.3 Command Line-Based Configuration
In certain environments, it may not be convenient to configure the JXTA platform via a graphical interface. In that case, you must configure the environment by hand. There are two issues here: the network configuration and the security configuration.

The network configuration information is stored within the `PlatformConfig` file located in the current directory. If you want to change network parameters, you can edit this file by hand before starting your JXTA application. If you are running a brand new JXTA application, you can copy an existing `PlatformConfig` file, make any necessary changes to it, and then run your application.

Alternately, if no windowing system is available when the JXTA application starts, it will prompt you as to whether it should continue:

**The window-based configurator does not seem to be usable.**

Do you want to stop and edit the current configuration? [no]: Using the current configuration.

If you answer this question with anything other than the string "yes," JXTA will create a default `PlatformConfig` file and run with the network parameters set in that file. If you do answer "yes," the application will throw an exception and exit, allowing you to edit the `PlatformConfig` file.

You must configure the username and password differently. The username and password information is stored within the `pse` directory. If that directory does not exist, then the platform will always attempt to run the configurator. If the configurator cannot run, you will get the same prompt as above, but there is no default username or password, and the application will ultimately fail.

What you must do in this case is specify the desired username and password on the command line by setting these two properties:

```
-Dnet.jxta.tls.password=********
-Dnet.jxta.tls.principal=myusername
```

Note that the password must be specified in "the clear"; it will not echo as a series of asterisks.

Using these properties will initialize the correct entries in the `pse` directory and allow your application to continue.

The username and password must be specified every time a JXTA application runs; the second time you run the shell, you'll get a pop-up window asking you to specify the username and password. To avoid this pop-up window, you may specify these properties when you start the application.

There's an interesting difference here: if the application prompts you for the username and password, you will not be able to proceed unless you type in the correct values. If you specify the username and password via command-line properties, the application will continue to function until those values are needed. In most JXTA applications, the username and password are needed only to create secure (encrypted) pipes. Therefore, unless your application uses secure pipes, it will work perfectly even if you specify the incorrect username and password via the command line. We'll explore this a little bit further in Chapter 7.

### 2.6.4 Caching

The JXTA Shell caches information about peers it has discovered in the `.cm` directory. If you shut down a shell and then restart it, you'll notice that it still knows about the same peers, even before you execute a `peers -r` command. This caching feature is used by many JXTA applications, though it is not a requirement of JXTA: it's up to the application to determine how (or if) it should cache information about previously discovered peers.

So when the JXTA Shell restarts, it still knows about all of the previously discovered peers. But what happens if these peers have disconnected? In that case, you need to flush the list of peers. This is done with the `peers -f` command:

```shell
JXTA> peers
peer0: name = Test Shell 1
peer1: name = Test Shell 2
JXTA> peers -f
JXTA> peers
peer0: name = Test Shell 1
```
The point is that peers are strictly transient by nature, and you cannot assume that a peer will continue to exist if you restart a session (or even during the same session). Peers come, peers go, and it's up to JXTA applications to be prepared to deal with this fact.

The caching of peer information by the shell is a specific case of advertisement caching. Every network resource in JXTA is described by an advertisement. Advertisements are XML documents that publish the availability of a resource and specify how to access the resource (we'll discuss them in more detail later in this chapter). When we discovered a peer, what we really discovered was the advertisement for that peer.

Advertisements may be cached by peers. Caching advertisements facilitates advertisement discovery, as more peers are likely to cache the requested advertisement. Without caching, peers would have to crawl through the entire network to find the advertisement they are looking for. Thus, caching reduces the exponential growth of messages sent within the network to discover a resource.

Each advertisement contains a time-to-live value that specifies the lifetime of the advertisement in the JXTA network. The lifetime is specified as a time relative to when the advertisement was published; when the lifetime expires, the advertisement is purged from the cache. This enables the network to maintain itself without centralized control.

As we mentioned, the JXTA specification does not specify how peers cache information; each peer is free to use a caching policy appropriate for its own environment. A PDA may cache very few advertisements only for its own usage. An enterprise server may cache more advertisements and supply them to other peers in the network. However, the standard JXTA API bindings that we discuss in later chapters automatically perform this caching. So by default, most JXTA applications will cache their advertisements.

2.7 Pipes

One of the most powerful of JXTA’s services is the pipe service. The idea of a pipe is familiar to users of Unix systems: a pipe is used to connect the output from one command to the input of another command. On a Unix system, if you want to count the number of unique exceptions that occur in a log file, you might use this command:

```
piccolo% cat log | grep "Exception" | sort | uniq | wc -l
```

The `cat` command prints the `log` file to its standard output. The `grep` command reads this file from its standard input and searches for lines containing the string "Exception"; it prints matching lines to its standard output. The `sort` command reads these lines from its standard input and sends the sorted list to its standard output, where it is read by the `uniq` command, which removes duplicate lines. The unduplicated lines are sent to its standard output, where they are read by the `wc` command, which counts the number of lines and finally prints that number.

Pipes are quite useful in that they allow you to build complex functionality from a number of simple building blocks. JXTA takes the familiar idea of pipes and extends their functionality to the network.

2.7.1 Peer Endpoints

JXTA pipes are defined in terms of the endpoints available to a peer. A peer endpoint is a logical abstraction of an address on a network transport that is capable of sending and receiving network messages. In the examples we’ve seen so far, the network transport has always been IP-based: the shell peer we’ve looked at has a TCP endpoint (port 9701 by default) and can have an HTTP endpoint. However, JXTA does not make this assumption and allows an endpoint to be an address on any network transport as long as the network is capable of sending and receiving datagram-style (i.e., unreliable packet-based) messages.

In the shell, the available endpoints are established when the configurator runs. The TCP endpoint is based on the network address selected from the pull-down menu and the port number entered into the TCP port text box; the HTTP endpoint is enabled by selecting support for HTTP.

2.7.2 Pipe Abstraction
In JXTA, pipes provide a unidirectional, virtual connection between two pipe endpoints: input pipe (receiving end) and output pipe (sending end). Pipe connections are established independently of the pipe endpoints peer location. For example, the input pipe endpoint can be located behind a firewall or NAT while the output endpoint can be located on a peer on the Internet. The endpoints may even be on physically different networks: the input pipe endpoint could be on a TCP network while the output pipe endpoint is on a token ring network. As long as there are available JXTA relay peers between the two endpoints, a logical pipe between them may be defined.

Therefore, pipes enable connections without any consideration of connectivity. Two peers may require an intermediary routing peer to communicate between each other. Pipes virtualize peer connections to homogenize and provide an abstraction of the full connectivity available within the JXTA network.

Pipe connections are layered on top of the peer endpoint connections, as we show in Figure 2-4. This figure shows a set of peer endpoint connections: Peer B, for example, has two HTTP endpoint connections while Peer D has two HTTP endpoint connections and one TCP/IP endpoint connection.

**Figure 2-4. Pipe connections over peer endpoints**

Note that some peers sit behind a firewall. HTTP proxy connections are used to connect to a peer outside the firewall (a TCP/IP Socks connection could also have been used). Peer C and Peer A can act as router peers for peers behind firewalls.

A pipe connection may involve multiple peer endpoint transport connections (multi-hops). The pipe connection between Peer B and Peer E involves an HTTP connection between Peer B and Peer C, an HTTP connection between Peer C and Peer D, and a TCP/IP connection between Peer D and Peer E. On the other hand, if a pipe connection is established between Peer A and Peer C, a single peer endpoint connection will implement the pipe, as Peer A and Peer C have a direct TCP/IP connection.

Pipes may send messages in different ways, each of which may provide a different quality of service. Some examples of this include:

- **Unidirectional, asynchronous**
  - The pipe endpoint sends a message, and no guarantee of delivery is made.

- **Synchronous request/response (RPC)**
  - The pipe endpoint sends a message and receives a correlated answer.

- **Publish/subscribe**
  - A pipe endpoint subscribes to messages sent from a publisher endpoint.

- **Bulk data transfer**
  - The pipe provides reliable data transfer of binary data.

- **Streaming**
  - The pipe provides efficient data transfer over a flow-controlled channel.

The default service provided by a JXTA pipe is unidirectional, asynchronous communication. All other pipe services can be implemented by the developer either directly on top of the peer
endpoints or over the default pipe abstraction. The unidirectional and asynchronous pipe service was selected as the minimum common denominator service easily implementable on a wide variety of non-IP transports. The asynchronous model was also selected since it fits well with the unreliable nature of a P2P network such as JXTA, in which peers or connections may disappear at any moment. Asynchronous models are also known to be more scalable (as the number of peers increase) than synchronous models. Finally, an asynchronous model provides an easier fault recovery and garbage collection mechanism when network connections fail.

2.7.3 Nonlocalized Connections

Pipes provide a nonlocalized communication abstraction. Applications and services create pipe endpoints (an input pipe and an output pipe) to communicate independently of the pipe endpoints’ peer location. Due to the unreliability and the interchangeable nature of peers in the JXTA network, pipes provide a powerful mechanism to build fault-tolerant applications. When creating a connection to a service, the pipe abstraction permits the peer to bind the pipe endpoints dynamically to the best or more appropriate instance of the service independent of the peer location. This hides the location of a service peer from the application, which is extremely important on an unreliable network.

2.7.4 Fault-Tolerant Constructs

The indirection introduced by pipes also allows applications to bind to the most appropriate instance of a service: perhaps the closest service or the best performing service. JXTA presents a novel approach to address application reliability. By building and proliferating interchangeable services on peers, applications can adapt to the unreliable network environment by connecting to or changing to the most available or efficient service; the application can do this at any time and with no regard to the location of the service.

Pipes are an essential tool to build such services and applications. JXTA pipes introduce a fundamental network programming shift in which developers should not write applications that connect to a specific peer to access a unique service, but should write applications that discover the closest available service regardless of which peer is running the service.

2.7.5 Pipe Endpoint Binding

Pipe endpoints are dynamically bounded to a peer endpoint at runtime via the Pipe Binding Protocol (PBP). The pipe-binding process consists of searching for and connecting two or more pipe endpoints. When a message is sent over a pipe, it is sent by the local output pipe to the destination input pipe endpoint that is currently listening to the pipe. The set of currently listening peers (that is, the location of the input pipe endpoint) is resolved using the PBP.

Pipes are uniquely identified by a pipe advertisement. The pipe advertisement contains a unique pipe ID and an optional pipe name. Pipe advertisements are a resource within the JXTA network; they may be discovered just as we discovered peers and peergroups. Each pipe advertisement is associated with a unique pipe.

Applications use a pipe service to create pipe endpoints (both input and output) associated with a particular pipe advertisement. The pipe service uses pipe advertisements to identify the pipe and resolve the input pipe and output pipe endpoints.

---

\[2\] To send a message to an output pipe endpoint, the corresponding input pipe endpoint must have been created first. This is a limitation of the current JXTA v1.0 implementation; it is expected to be removed in the future.

2.7.6 Pipe Communication Mode

Pipes support two modes of communication (see Figure 2-5):

Point-to-point
A point-to-point pipe connects exactly two pipe endpoints: an input pipe receives messages sent from an output pipe. No reply or acknowledgment is supported. Additional information in the message payload (such as a unique ID) is required to determine the sequence of messages sent over the pipe. The message payload may also contain a pipe advertisement that can be used to open a pipe to reply to the sender.

**Propagate pipe**

A propagate pipe connects one output pipe to multiple input pipes. Messages flow into the input pipes from the output pipe (propagation source). A message sent over a propagate pipe is sent to all listening input pipes; this process may create multiple copies of the message. On a TCP/IP network, IP multicasting is used as an implementation for propagate pipes when the propagate scope maps to an underlying physical subnet in a one-to-one fashion. Propagate pipes can also be implemented using point-to-point communication on transports that do not support multicasting.

**Figure 2-5. Pipe communication modes**

2.7.7 Pipes and Peergroups

Pipe connectivity is related to the concept of peergroups: only pipe endpoints that are located in the same peergroup can be mutually resolved by a pipe service. Each peergroup has its own pipe service, so to open a pipe connection between two peers, the two peers must have joined the same peergroup.

Of course, since all peers are part of the NetPeerGroup, any peer can open a pipe to any other peer. The difference is in which pipe service will resolve the pipe. In Figure 2.5, Peer 1 can use the pipe service of either the NetPeerGroup or of Peergroup A to resolve its pipe connections to Peers 2 and 3. Peers 2 and 4 can use only the pipe service of the NetPeerGroup to resolve their mutual pipe connection. The context of the pipe resolution is important because of the security context that may be enforced by the pipe service of a peergroup: Peer 1 may decide that it does not want to send data to any peer that has not been authenticated into Peergroup A. It then advertises only its pipe endpoint within that peergroup.

A peer can maintain different pipe connections to the same peer, holding each of them in a different peergroup context for security reasons. Messages can be sent to the peer with different security levels depending on the pipe used.

2.7.8 Pipe Examples

Pipes are used behind the scenes for many shell services. One such service is the talk service, which allows users in two different shells to send simple string messages to each other. To use the talk service, you must register two users (either in the same or — ideally — different shells). One registration looks like this:

```
JXTA> talk -register sdo
.......  
User: sdo is now registered
```
JXTA> **talk -login sdo**
Now repeat the process in a second shell with a different user:
JXTA> **talk -register jra**
......
User: jra is now registered
JXTA> **talk -login jra**
In the first shell, you can then send a message from sdo to jra:
JXTA> **talk -u sdo jra**
found user’s advertisement attempting to connect
talk is connected to user jra
Type your message. To exit, type "." at beginning of line
Hello!
In the second shell, you’ll see this message:
talk: from sdo to jra
Message: Hello!
Behind the scenes, this example uses a number of JXTA services; the one that concerns us for now is the pipe service. The message from sdo to jra is sent via a pipe. What happens is this:

- A user is registered. This creates an advertisement that the peer will accept talk messages.
- The user logs in. This creates the actual input pipe. A thread is set up to read continually from the input pipe, which is why the message from sdo to jra appeared asynchronously in the second shell window. This is why logging in is a necessary step in this service; without it, there would be no input pipe to accept messages.
- The user sends a message. This creates an output pipe; whatever data is written to this output pipe will be read on the input pipe of the user to whom the message is directed.

We can explore some other shell commands to understand a little more about how JXTA pipes work. To create a pipe, we must make a pipe advertisement and use it to create the pipes:
JXTA> **pipeadv = mkadv -p**
JXTA> **inpipe = mkpipe -i pipeadv**
JXTA> **outpipe = mkpipe -o pipeadv**
This creates both an input and output pipe. In order for the pipes to be connected to each other, they must be created with the same advertisement that we created here. In the talk service (and in other JXTA applications), you discover input pipe advertisements that are created by the shell in response to the talk -register command. The shell uses this pipe advertisement to create the input pipe in response to the talk -login command; it uses this pipe advertisement to create the output pipe in response to the talk user command.
In this example, we’ve created the input and output pipes in the same peer. Later, we’ll see how to create the pipes in different peers.

### 2.7.9 Messages

The information transmitted through pipes is **messages**. Messages define an envelope to transfer any kind of data: text, code, and so on. Furthermore, a message may contain an arbitrary number of uniquely named **sections**. Each section has an associated MIME type and can hold any form of data. Binary data may be encoded using the Base64 encoding scheme in the body of a section; a CDATA section may also be used. Some sections may contain XML-structured documents. Applications and services communicate by constructing messages and sending and/or receiving messages through the input and output pipe endpoints.

JXTA messages use a binary format to enable the efficient transfer of binary and XML data. A JXTA binary message format is composed of a sequence of **elements**. Each element has a name and a MIME type and can contain either binary or XML data. The form of a JXTA message is shown in **Figure 2-6**.

**Figure 2-6. A JXTA message**
In order to send data over a pipe (or to any JXTA peer), the data must be encapsulated in a message. In the shell, this can be done by importing an existing file that contains the message body and using the `mkmsg` command to convert the data into a message. When this is done, the imported data will be associated in the message with a tag of your choosing.

First, we need to create a file containing the message data. Such a file would contain a set of arbitrary XML tags; a simple example is the file containing the single line:

```
<Data>Hello, world!</Data>
```

If this file is named `data`, then we can import it into the shell like this:

```
JXTA> importfile -f data mydata
```

The `mydata` variable now contains the data read in from the file. In fact, it contains other information as well, since the shell has embedded some additional XML into it.

You can create the actual message like this:

```
JXTA> mymsg = mkmsg
JXTA> put mymsg mytag mydata
```

The first command creates an empty message named `mymsg`; the second populates this message with the data from the `mydata` object, associating it with the `mytag` tag.

If you've made the pipes and message, you can then send and receive data like this:

```
JXTA> send outpipe mymsg
JXTA> newmsg = recv inpipe
```

`recv` has received a message

The message is sent on the output pipe. The input pipe then reads the message and stores it in a new message variable, `newmsg`. The contents of `newmsg` are the same as those of `mymsg`; they have a tag of `mytag` and an associated message body that contains the structured XML document created by reading the datafile. The message body can be extracted with the `get` command:

```
JXTA> newdata = get newmsg mytag
JXTA> cat newdata
```

```
<?xml version="1.0">
<ShellDoc>
  <Item>
    <Data>Hello, world!</Data>
  </Item>
</ShellDoc>
```

## 2.8 Advertisements

We've mentioned advertisements in passing a number of times in this chapter; we even created advertisements for pipes and groups using the shell's `mkadv` command. Advertisements are one of the basic building blocks of JXTA: they help peers discover any kind of service, including peergroups, other peers, and pipes. In fact, at a basic level, the JXTA infrastructure is simply about sending advertisements for various resources around to interested parties. A JXTA application simply searches for advertisements it is interested in and responds to requests for advertisements that it has published. The basic protocols we've mentioned all use advertisements as the mechanism by which they send data. Advertisements provide a platform-independent representation of platform objects that can be exchanged between different platform implementations (Java, C, etc.).
Advertisements are structured XML documents. The JXTA infrastructure defines six such documents:

- Peer advertisement
- Peergroup advertisement
- Pipe advertisement
- Service advertisement
- Content advertisement
- Endpoint advertisement

Developers may subclass these advertisements when they create their own services. For example, a content advertisement could announce that a peer has a particular item (such as a PDF file of this chapter). A JXTA application may create a subclass of the content advertisement to refer only to PDF files of this book. Peers who are interested in generic content can look for standard content advertisements; peers who are interested in only the PDF files of this book can look for the content subclass. Both peers will find the PDF of this chapter, but the second will find less extraneous content.

2.8.1 Obtaining Advertisements

There are three ways to obtain advertisements. First, peers create advertisements for services they are interested in supplying. For example, in the previous section we used the `mkadv -p` command to create an advertisement for a peer endpoint. Second, peers discover some kinds of advertisements. Advertisements for other peers, for instance, are broadcast through a network, where peers will discover them. Finally, peers can search for advertisements: they can query other peers for advertisements that have specific attributes.

In our previous pipe example, we set up both ends of the pipe within the same peer. Now we’ll see how advertisements allow us to set up the ends of the pipe in different peers. To follow along with this example, you’ll need to start two shells (preferably on different machines).

First, one peer must create the pipe advertisement. In the first shell, execute this command:

```
JXTA> pipeadv = mkadv -p
```

Now this advertisement must be published—i.e., made available to other peers. In the first shell, this is done with the `share` command:

```
JXTA> share pipeadv
```

Other peers must obtain this pipe advertisement. In the second shell, this is done with the `search` command:

```
JXTA> search -r
JXTA> search
```

```
JXTA Advertisement adv0
JXTA Advertisement adv1
JXTA Advertisement adv2
JXTA Advertisement adv3
```

The first command tells the shell to search for advertisements using discovery (e.g., by broadcasting a message to other peers in the peergroup in order to see what shared advertisements they have). The second command prints out a list of all the advertisements that the peer has obtained (advertisements that the peer itself created will also appear in this list). Note that it may take a while between when the `search -r` command is issued and when the remote advertisement is discovered; you may need to wait a minute or two before the desired advertisement shows up in the list.

You may also see multiple advertisements listed; in order to determine which is the desired pipe advertisement, you must `cat` each advertisement. The pipe advertisement we’re interested in looks something like this:

```
JXTA> cat adv3
<?xml version="1.0"?>
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org?">
If you `cat` the `pipeadv` variable in the first shell, you can determine the ID to look for in the list of advertisements in the second shell.

Now we can set up the actual pipes. In the first shell, set up the input pipe:

```bash
JXTA> inpipe = mkpipe -i pipeadv
```

In the second shell, set up the output pipe:

```bash
JXTA> outpipe = mkpipe -o adv3
```

Note that in the first shell, we used the `pipeadv` variable to create the pipe; in the second shell, we must use the variable that the shell assigned when it discovered the pipe advertisement.

At this point, you can create a message in the second shell and use the `send` command to deliver the message to the first shell (which reads it using the `recv` command). This procedure is unchanged from our previous example.

## 2.9 Summary

In this chapter, we used the JXTA Shell to explore the underpinnings of the JXTA framework. Each instance of the shell is a peer, and we've seen how multiple peers can use a variety of techniques to discover other peers. As implemented in the shell, these techniques are tied to a particular network binding (e.g., TCP), though the PDP is designed to run over any network.

Similarly, we've seen how the shell peers organize themselves into peergroups, another key JXTA concept. Peergroups allow for the segregation of services so that peers can more easily manage their relationships with other peers.

The communication between peers in JXTA is based on the notion of an endpoint. One key example of an endpoint is a pipe, which can provide either input or output for a peer and allows complex behavior to be built from several simple peers. Pipes form the basis of communication in JXTA; peers depend on advertisements to locate pipes and other endpoints with which they want to communicate.

The shell provides a mechanism by which we can explore and test many of these features. In the next few chapters, we'll look into how we can develop our own JXTA applications that include these same features.
Chapter 3. A Hello World Example

In the next few chapters, we’ll show you how to write JXTA applications using the Java language API provided with the JXTA platform. If you want to use a different language, the program structure that we describe will be helpful, though the syntax and API calls will be different. The JXTA API is used by application and service developers: it provides high-level constructs, such as pipes and peergroups, for application developers, as well as low-level constructs, such as advertisements and structured documents, for service developers. The examples in the next few chapters cover both types of constructs.

We’ll start in this chapter with a simple “Hello world” example. This example will cover the basic flow of control for a JXTA application and how a JXTA application is configured and executed. In later chapters, we’ll use a restaurant auction example to illustrate other key concepts, such as discovery, peergroups, pipes, advertisements, services, and security.

The JXTA platform provides an integrated environment to run applications. An application can be run in three ways:

- An application can be added to the JXTA Shell, after which it may be run interactively in an instance of the shell.
- A standalone application can be written to instantiate, configure, and use the JXTA platform.
- A JXTA service can be used by other JXTA applications. JXTA services can be started on demand.

In the next three chapters, we’ll focus on the second method. This method describes in more detail how an application interacts with the platform. We’ll discuss the last method in Chapter 6.

3.1 Peergroups

The first thing a JXTA application must do is discover and join the NetPeerGroup. As we explained in Chapter 2, every JXTA peer belongs to the NetPeerGroup (also called the World Peergroup). New peergroups are created within the NetPeerGroup and are composed of a subset of the NetPeerGroup peer members. From the NetPeerGroup, applications can discover and crawl through all other peergroups within the NetPeerGroup.

The NetPeerGroup permits all peers to find enough information to bootstrap and support themselves. For obvious scalability reasons, usage of the NetPeerGroup should be limited to bootstrapping. Peers should create and join other peergroups after bootstrapping themselves.

In the Java language, peergroups’ objects are instances of the PeerGroup class (net.jxta.peergroup.PeerGroup). There are ways to create your own peergroups, of course, but an object that represents the special NetPeerGroup is returned from the static newNetPeerGroup() method of the PeerGroupFactory class (net.jxta.peergroup.PeerGroupFactory). Therefore, the simplest of all JXTA applications looks like this:

```java
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.exception.PeerGroupException;

public class HelloWorld {
    static PeerGroup group = null;

    public static void main(String args[]) {
        HelloWorld myapp = new HelloWorld();
        myapp.startJxta();
        System.exit(0);
    }
}
private void startJxta() {
    try {
        // Create and start the default JXTA NetPeerGroup
        group = PeerGroupFactory.newNetPeerGroup();
    } catch (PeerGroupException e) {
        // Could not instantiate the group; print the
        stack and exit
        System.out.println("Fatal error: creating the net PeerGroup");
        System.exit(1);
    }
    System.out.println("Started Hello World");
}

When the newNetPeerGroup() method is called, the application will look for configuration
files in its current directory. If these configuration files are not found, the JXTA configurator will
run (as we saw in Chapter 2). After loading its configuration information, the application will use
the PDP to locate the NetPeerGroup; it will then join the NetPeerGroup.
If the NetPeerGroup cannot be located, then the application will automatically start a new
NetPeerGroup. This peergroup will be available on the local network and, as other JXTA
applications are started, they will use it as the NetPeerGroup.
From a protocol perspective, the newNetPeerGroup() method is quite complex. From a
developer's perspective, however, initializing the JXTA platform and joining the NetPeerGroup
requires one simple method invocation.

3.2 Running JXTA Applications

To compile this example, you need to add JXTA files to your classpath as we discussed in Chapter
2. Then you can simply compile the HelloWorld class:

piccolo% javac HelloWorld.java

The application is started as we might expect:

piccolo% java HelloWorld

As we mentioned, the first time you run the platform from a particular directory, the configurator
will pop up and ask you to configure the network environment. Subsequent invocations of the
application will require you to enter the secure username and password that you entered into the
configurator when you first ran the program. Once you've run the configurator and/or entered the
correct username and password, the platform starts booting, and the following message is printed:

Started Hello World

If you examine the directory in which the program was run, you'll discover that the .jxta directory
contains the platform configuration file (PlatformConfig), the peergroup cache directory (cm), and
the username directory (pse).

3.3 Advertisements

Let's look now at some JXTA advertisements. Advertisements are central to JXTA; you discover
JXTA peers and their resources by looking for advertisements, and you provide your own services
by publishing advertisements. Therefore, working with advertisements and the documents they
contain is the next step we must take in learning how to develop JXTA applications.
Advertisements are represented by the abstract Advertisement class
(net.jxta.document.Advertisement). Each advertisement has a unique type, which
is returned by the getAdvertisementType() method. The JXTA API defines the
following core advertisement types (each of which is an abstract class in the net.jxta.protocol package):

- PeerAdvertisement
- PeerGroupAdvertisement
- PipeAdvertisement
- PeerInfoAdvertisement
- EndpointAdvertisement
- ModuleClassAdvertisement, ModuleSpecAdvertisement, and ModuleImplAdvertisement (module-related advertisements)
- RdvAdvertisement
- TransportAdvertisement

Each advertisement object holds an object that implements the Document interface (net.jxta.document.Document) and represents the advertisement. An advertisement is typically represented as an XML document, and the JXTA protocol specification relies heavily on XML advertisements. The getDocument( ) method of the Advertisement class returns a representation of the advertisement in different formats based on its MimeMediaType argument. Typical MIME types are "text/xml" or "text/plain" (which, in fact, are the only two advertisement types presently represented by JXTA).

### 3.3.1 The Document Interface

JXTA documents are simple data containers. JXTA does not interpret any of the data contained in a Document object. Each document is typed using a MIME type (an object of the net.jxta.document.MimeMediaType class). Therefore, the Document interface is somewhat analogous to an HTTP stream: no encoding or compression information is specified except by the MIME information associated with the data.

#### 3.3.1.1 Structured document interfaces

An important subinterface of the Document interface is the StructuredDocument interface (jxta.net.document.StructuredDocument). The StructuredDocument interface defines a document that is composed of a hierarchy of elements (each of which implements the jxta.net.document.Element interface); this works very much like an XML document. The StructuredDocument interface allows the content of several document types to be manipulated in an abstract way without regard to their physical representation. Structured documents are one of the elementary content types that are manipulated by the core JXTA APIs. The JXTA API also defines a StructuredTextDocument interface (jxta.net.document.StructuredTextDocument) that extends the StructuredDocument interface. This interface is convenient for manipulating text-based documents, such as XML documents. While the StructuredDocument interface manipulates elements as abstract objects, the StructuredTextDocument interfaces manipulates the keys and values of elements as strings.

### 3.3.2 Working with Advertisements

There are two ways to obtain an Advertisement object. You can, of course, create advertisement objects. This is necessary when publishing the advertisement (we’ll do this in later chapters). In addition, JXTA objects that represent resources usually contain their advertisement as a property; you can retrieve the advertisement used to create a peergroup via the getPeerGroupAdvertisement( ) method of the PeerGroup class.
We can modify our previous HelloWorld example to display more information about the advertisement that was used to create the NetPeerGroup:

```java
private void startJxta() {
    try {
        // Create and start the default JXTA NetPeerGroup
        group = PeerGroupFactory.newNetPeerGroup();
    } catch (PeerGroupException e) {
        // Could not instantiate the group; print the stack and exit
        System.out.println("Fatal error : creating the net PeerGroup");
        System.exit(1);
    }

    System.out.println("Started Hello World");

    // Get the PeerGroup advertisement
    pgadv = group.getPeerGroupAdvertisement();
    try {
        // Print out the PeerGroup advertisement document. To do so, we must convert it to a text document with whatever MIME type we want.
        // First, we’ll print it as a plain-text document.
        StructuredTextDocument doc = (StructuredTextDocument) pgadv.getDocument(new MimeMediaType("text/plain"));

        // Now that we have a plain-text document, we can print the document
        // via the StructuredTextDocumentsendToWriter method
        StringWriter out = new StringWriter();
        doc.sendToWriter(out);
        System.out.println(out.toString());
        out.close();

        // For comparison, we’ll do the same thing but print it
        // out as an XML document
        StructuredTextDocument adoc = (StructuredTextDocument) pgadv.getDocument(new MimeMediaType("text/xml"));
        StringWriter aout = new StringWriter();
        adoc.sendToWriter(aout);
        System.out.println(aout.toString());
        aout.close();
    } catch (Exception ex) {
        ex.printStackTrace();
    }
}
```

Here, we use the `getDocument()` method to obtain the structured text from the advertisement passed to the `init()` method. We first ask for a document with the MIME type.
"text/plain," which will convert the document into plain text. We then use the "text/xml" type and receive the actual XML document; we’ll see the difference in the output.

To print the advertisement document, we use the sendToWriter( ) method of the TextDocument class. We create a StringWriter object to use as the output stream and then send the contents of that writer to System.out. We’ll use this construct frequently in future examples.

If we run the new version of the HelloWorld class, we get the following additional output:

```java
jxta:PGA :
   GID : urn:jxta:jxta-NetGroup
   MSID : urn:jxta:uuid-DEADBEEFDEAFBABAFAEDBABE000000010206
   Name : NetPeerGroup
   Desc : NetPeerGroup by default

<?xml version="1.0"?>
<!DOCTYPE jxta:PGA>
<jxta:PGA xmlns:jxta="http://jxta.org">
   <GID>
      urn:jxta:jxta-NetGroup
   </GID>
   <MSID>
      urn:jxta:uuid-DEADBEEFDEAFBABAFAEDBABE000000010206
   </MSID>
   <Name>
      NetPeerGroup
   </Name>
   <Desc>
      NetPeerGroup by default
   </Desc>
</jxta:PGA>
```

The output of the advertisement has been converted into both plain and XML text. In the plain text, you can see that some structure of the advertisement has been preserved: the tags of the original advertisement are all represented as key/value pairs separated by a colon.

### 3.3.3 Building Advertisements

In later chapters, we’ll need to build several different types of advertisements. Regardless of their type, all advertisements are created in the same way. First, a blank advertisement document is retrieved from the AdvertisementFactory class (net.jxta.document.AdvertisementFactory). Then, individual elements within the advertisement document must be set using methods within the specific type of advertisement. For example, here’s how we build a module specification advertisement:

```java
 ModuleSpecAdvertisement msa = (ModuleSpecAdvertisement) AdvertisementFactory.newAdvertisement( ModuleSpecAdvertisement.getAdvertisementType( ));
 msa.setName("JXTASPEC:RMIService:HelloService");
 msa.setVersion("Version 1.0");
 msa.setCreator("sun.com");
 msa.setSpecURI("http://www.jxta.org/tutorial/RMIService.jar");
 msa.setModuleSpecID(IDFactory.newModuleSpecID(mcID));
 Element e = doc.createElement("Stub", stub);
```
Like all advertisements, module specification advertisements are built by calling the `newAdvertisement()` method of the `AdvertisementFactory` class. Each specific advertisement class that we listed earlier has a `getAdvertisementType()` method that the factory uses to learn which type of advertisement to instantiate. After creating the advertisement, you must set every attribute within the advertisement; consult Chapter 14 to see the appropriate attributes to set for each advertisement type.

In this case, we’ve set the name of the advertisement, its version and creator, the location from which the code that implements the module can be located, the ID of the module that we’re advertising, and an additional document containing parameters that are used to create the module. As we’ll see in later examples, certain services have methods that allow us to retrieve a fully populated advertisement. In this case, setting all the attributes of the advertisement is not required.

### 3.4 Peergroup Services

All peergroup objects carry certain information: the name of the peergroup, its unique universal identifier (UUID), and so on. In addition, peergroups provide certain core JXTA services, as we mentioned in Chapter 2. In particular, peergroups can provide a discovery service that can be used to discover other JXTA peers and services, a pipe service that can be used to create input and output pipes, and a membership service that can be used to join the peergroup. All of this information can be retrieved from the peergroup object:

```java
private void startJxta() {
    try {
        // Create and start the default JXTA NetPeerGroup
        group = PeerGroupFactory.newNetPeerGroup();
        } catch (PeerGroupException e) {
            // Could not instantiate the group; print the stack
            System.out.println("Fatal error : creating the net PeerGroup");
            System.exit(1);
        }
    
    System.out.println("Started Hello World");
    
    // Now we’ll access the peergroup service
    // and get various information from it
    PeerGroupID pgid = group.getPeerGroupID();
    System.out.println("pgid= " + pgid);
    PeerID pid = group.getPeerID();
    System.out.println("pid= " + pid);
    String name = group.getPeerName();
    System.out.println("peer name= " + name);
    
    // Get the core services. We don’t use the core services in this
    // example, but this is how you retrieve them if you need them.
    DiscoveryService disco = group.getDiscoveryService();
    PipeService pipe = group.getPipeService();
```
MembershipService member = group.getMembershipService();
ResolverService resolv = group.getResolverService();
System.out.println("All done");

} catch (Exception ex) {
    ex.printStackTrace();
}

The information printed by this code looks like this:
Started Hello World
pgid= urn:jxta:jxta-NetGroup
pid= urn:jxta:uuid-59616261646162614A7874615032503368D0F1C1B8B54A3ABE2F8D
DEEE47489F03
peer name=Test Application 1
All done

The peer name ("Test Application 1") is the name you entered when the configurator ran. We'll see how the services are used in later chapters.

3.5 Summary

In this chapter, we provided the skeleton of a basic JXTA application. This application shows us how to initialize the JXTA platform and perform basic operations with JXTA advertisements and peergroups.
While the HelloWorld class is a complete JXTA application and has allowed us to understand the basic structure of programming in JXTA, it doesn’t actually do anything except print out a lot of interesting information. In the next few chapters, we’ll develop a more complete example, one that leverages the features of the JXTA platform to create a community of cooperating peers.
Chapter 4. Service Advertisement and Discovery

In the next few chapters, we'll expand on the JXTA application interface and develop a complete JXTA application that is both a decentralized service and an application that uses the service. The complete example is spread out over a few chapters; we begin in this chapter by showing how to handle service advertisement and discovery. In later chapters, we'll add the pipe communication between the service and application as well as handling some security issues.

4.1 An Auctioning Example

The example we'll build will perform auctioning. Auctioning applications are great for demonstrating decentralized, P2P interactions between auctioneers and bidders. The JXTA programming model can be applied with great simplicity to build a cost-effective and efficient auctioning system.

4.1.1 The Client/Server Auctioning Model

Auctioning systems are typically implemented with a centralized (client/server) auctioning server model (see Figure 4-1). The central auctioning server receives auction offers and bids, and matches them. While centralized auction architecture has some advantages, it presents some significant scalability, security, and reliability issues. The auctioning server needs to be scaled when the number of auctions and bid requests increase, as performance tends to decrease as more and more users are active on the system.

Figure 4-1. The client/server auction architecture

The auctioning server is a single point of failure for the entire system: if the server goes down, all transactions are stopped. While clustering and failover technologies are typically used to address single server failures, there is a tremendous cost associated with maintaining and growing a centralized server infrastructure. One cannot expect to continue to grow centralized server computing facilities without reaching a point at which too much complexity is introduced. Furthermore, as these facilities become larger, their level of unreliability is increased because more components can fail. Auctioning servers are also the perfect target for denial of service attacks as intruders and attackers can focus their effort on a limited set of servers to bring the entire system down.

Since all transactions are processed on the auctioning server, user information can be easily logged by the auctioning provider, potentially invading the privacy of the auctioneers or bidders. Auctioneer profiles can be easily captured and sold to marketing research companies. Sensitive user information (email or personal contact information) is typically maintained on the auctioning server, creating the possibility of an intruder accessing this information.
4.1.2 The P2P Auctioning Model

In a JXTA P2P model, things are different. When sellers and bidders have found each other, they do not need an intermediary to broker their transactions. Sellers and bidders can interact directly to finalize the transaction.

Using a P2P architecture for building an auctioning system provides a number of advantages:

**Scalability**
As the number of users increase in the system, performance will tend to increase. More peers will be available to participate and help other peers find new bids and auctions.

**Reliability**
The system has no single point of failure. Sellers and bidders can find each other even if a set of peers is down. A bidder can always bid on another auction if a previous seller is not responding.

**Security**
No centralized user database needs to be maintained; each peer can maintain its own private information so that it is not accessible by any other peer. Since transactions occur directly between peers, it is difficult to trace and record user patterns.

**Robustness**
Since all information is decentralized, it is very difficult to perform a denial of service attack. Every peer is interchangeable; if one peer is attacked, users can go to another peer.

In the next few chapters, we’ll develop an auctioning system within the JXTA P2P model. We want to create an auction service in which restaurants can advertise their food to hungry people who want to bid for it. The architecture of this example is shown in [Figure 4-2](#). In this architecture, there are the following entities:

**The peergroup (RestoNet)**
The RestoNet peergroup defines the scope of interaction and discovery between all peer entities involved in the restaurant auction. It is composed of all the peers that participate in the auction. Peers must join this peergroup to participate in the auction as either restaurant sellers or bidders.

**Restaurants (RestoPeers)**
Restaurants provide dining services. They discover and join the RestoNet peergroup, advertise their food services, and receive bids for their services. Restaurants respond to the dining auctions they can satisfy.

**Hungry individuals (HungryPeers)**
Hungry individuals are searching for a place to eat. They will join the RestoNet peergroup in order to discover restaurants. They will then publish dining requests and wait for a possible bid from one or more restaurants. Individuals will typically select restaurants depending on a number of criteria (price, food type, distance, and time); they are typically looking for the best offer and the best food.

[Figure 4-2. The P2P auctioning architecture](#)
While a P2P architecture such as JXTA provides many advantages in the implementation of an auctioning system, some problems must still be addressed. Since there are no centralized services, auctioneers and bidders need to find each other without relying on costly network-crawling techniques. Lost connectivity between certain peers may partition the network, precluding one side of the network from performing transactions with the other side. Usage of the system is difficult to track and monitor. These problems can be addressed in many ways, resulting in an efficient P2P auctioning system. In our examples, we will show how JXTA features help.

4.2 JXTA Discovery

The first thing a peer must do is discover its surrounding environment. Peers discover resources, support themselves, access needed services, and expand their horizons in their surrounding environments. In the case of a RestoPeer, the first thing it must do is discover and join the RestoNet peer group. There is, of course, no guarantee that the RestoNet peer group already exists, so the RestoPeer must be prepared to create it as well.

JXTA discovery is closely related to the peer group concept: discovery in JXTA is performed within the context of a peer group. We have seen that every peer joins the NetPeerGroup. Discovery performed within the context of the NetPeerGroup enables any peer to potentially discover any advertisements published in the NetPeerGroup. Due to the large number of advertisements in the NetPeerGroup, it is impractical to perform discovery within the context of the NetPeerGroup; peers should attempt to discover peer groups within the NetPeerGroup context and then attempt to discover other resources within a smaller peer group.

4.2.1 Discovery Scope

The discovery scope of a peer corresponds to the peer group context in which discovery is performed. For example, in Figure 4-3, if discovery is performed within the context of Peer group 2, only advertisements stored on peers that are members of the Peer group 2 can be discovered. The peer group context enables a peer to scope discovery operations and reduces the risk of overwhelming the network.

Figure 4-3. The discovery scope of JXTA peer groups

Note that a peer may belong to multiple peer groups (shown as intersecting peer groups in Figure 4-3). When publishing advertisements, the publishing peer defines the scope of discovery by selecting which peer group the advertisement is published in. The same advertisement may be published in multiple peer groups. In Figure 4-3, Peer A publishes an advertisement in both Peer group 2 and Peer group 3. This means that only members of Peer groups 2 and 3 can discover the advertisement; members of Peer group 1 cannot discover the advertisement. Each peer group corresponds to a different discovery scope.

4.2.2 The Asynchronous Discovery Model
The JXTA discovery API uses an asynchronous mechanism for discovering advertisements. A peer that wants to discover a resource propagates a discovery request within a chosen peergroup scope. Responses are received asynchronously. No guarantee is made regarding when (or if) a response will be received: multiple responses may be received, or no responses may be received. Peers may have cached advertisements that represent resources that are no longer reachable: the peer itself may be down, or no routes to the peer may be available. A discovery request may be sent using endpoint transports with different latencies, leading to unpredictable behaviors. The unpredictability and unreliability of a P2P network such as JXTA makes it difficult to specify time-out or predictable behaviors. It is ultimately the responsibility of an application to decide how long it should wait to get a response.

An asynchronous programming model is best suited for this unreliable P2P network environment. Asynchronicity greatly simplifies error recovery, since errors are handled as normal cases rather than as exceptions.

A peer that wants to discover a particular advertisement must specify the following:

**Type**

The type of advertisement that the peer wants to discover (e.g., a peergroup advertisement or a peer advertisement). As we’ve mentioned, there are core advertisement types (e.g., peer, peergroup, pipe, and so on); the type itself is based on the root element of the XML document that comprises the advertisement.

**Attribute and value**

The advertisement must contain an XML tag that corresponds to the given attribute and contains the given value. For example, the service advertisement for the rendezvous service contains an XML tag: `<Name>jxta.service.rendezvous</Name>`. So a peer that wants to discover a rendezvous service would supply an attribute of Name and a value of `jxta.service.rendezvous`.

**Threshold**

Some methods of the discovery API allow a peer to specify a threshold that defines an upper limit for the number of matching advertisements that should be returned to the requesting peer. This allows the peer to control the size of the responses it will receive; small memory devices may be able to store only a few advertisements in memory.

Advertisements are cached in local storage by a peer; as they are asynchronously discovered, a background thread automatically places them in the cache. Therefore, even though discovery is an asynchronous process, the developer does not need to worry about threads or any other asynchronous technique; she simply sends the discovery request and later checks back to see if there were any responses.\(^1\)

\(^1\) Though this is the simplest way to handle discovery, there is a notification-based system available as well, which we discuss in Chapter 5.

The cache of previously discovered advertisements is often held in persistent storage; this is certainly the case with the Java 1.0 JXTA implementation for desktop systems and servers, though other implementations (particularly on small devices) may not have this option. Therefore, an application that looks for responses in the local cache may find responses that were cached from its previous runs as well as any new ones it finds while running.

All discovery operations are handled by the discovery service, one of the core JXTA services. As we saw in Chapter 3, a handle to the discovery service may be obtained by calling the `getDiscovery()` method of the peergroup object with which the application is initialized; this method returns a `Discovery` object (`net.jxta.discovery.Discovery`). This class has two main methods for discovering advertisements:

```java
public Enumeration getLocalAdvertisements(int type, String attribute, String value) throws IOException
```

This method is used to search the local cache for already discovered advertisements. Even though there are six core advertisement types, the type passed to this method must be one of these three values: `PEER` (peer advertisement), `PEERGROUP` (peergroup advertisement), or `ADV` (all other advertisements).

```java
public void getRemoteAdvertisements(String peerid, int type, String attribute, String value, int threshold)
```
This method is used to send a discovery request to members of the peergroup. Remember that in most cases the discovery request will not be able to reach all members: some peers may be down or not reachable. It is also not practical or realistic to expect all peers to respond to a discovery request.

If a peer ID is passed to this method, only that peer will be contacted, and only the matching advertisements known to that peer will be returned; this is often used with rendezvous peers.

If null is passed as the peer ID, then the peer will attempt to contact all known rendezvous peers. In addition, any broadcasting capabilities available in the underlying endpoint transports will be used to propagate a request. In the case of TCP/IP, IP multicast is used to propagate a request to all member peers in the same subnet. This approach limits the scope of discovery to a subnet, as most routers are configured to block multicast packets. HTTP does not have a propagate capability.

Let’s apply all of this to the restaurant auctioning example. We’ll first look at the RestoPeer class.

## 4.3 The Service Implementation

Here’s our first implementation of the RestoPeer class (the code that provides the restaurant service). A RestoPeer is responsible for discovering and joining the RestoNet peergroup; if the peer cannot find the RestoNet peergroup, the peer will create it.

The RestoPeer class has a simple structure:

1. Discover the existing RestoNet peergroup advertisement.
   a. Look for any previously discovered advertisements in the local cache.
   b. Send the PDP request to discover the advertisement.

   This may be done using a broadcast-type mechanism or by contacting known rendezvous peers, either of which is a synchronous call to the developer. Behind the scenes, the API will create a new thread that will listen for responses. As these responses come back, the background thread will read them and insert the responses into the peer’s local cache of advertisements.

   c. Wait a while for any responses to come back.
   d. Look again for responses in the local cache.

2. If the existing RestoNet is not found, create a peergroup advertisement for it. Then create the peergroup based on that advertisement.

3. If the existing RestoNet is found, use the discovered advertisement to join the peergroup.

Here’s the code necessary to perform these steps:

```java
import java.io.*;
import java.net.*;
import java.util.*;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.exception.PeerGroupException;
import net.jxta.discovery.DiscoveryService;
import net.jxta.id.IDFactory;
import net.jxta.pipe.PipeService;
import net.jxta.protocol.PeerGroupAdvertisement;
import net.jxta.protocol.ModuleImplAdvertisement;
```
// RestoPeer represents a restaurant that receives auction requests
// for food from HungryPeers. The RestoPeer will discover and join
// the RestoNet and publish itself as a provider for HungryPeers

public class RestoPeer {

    private PeerGroup netpg = null; // The NetPeerGroup
    private PeerGroup restoNet = null; // The RestoNet
    private String brand = "Chez JXTA"; // Brand of my restaurants
    private int timeout = 3000; // Time-out; can be adjusted

    // Services within the RestoNet peer group
    private DiscoveryService disco; // Discovery service
    private PipeService pipes; // Pipe service

    static String groupURL = "jxta:uuid-4d6172676572696e204272756e6f202002";

    public static void main(String args[]) {
        RestoPeer myapp = new RestoPeer(  );
        myapp.startJxta(  );
        System.exit(0);
    }

    // Start the JXTA application
    private void startJxta(  ) {
        try {
            // Discover and join (or start) the default peer group
            netpg = PeerGroupFactory.newNetPeerGroup(  );
        } catch (PeerGroupException e) {
            // Couldn’t initialize; can’t continue
            System.out.println("Fatal error : creating the net PeerGroup:);
            System.exit(1);
        }

        // Discover (or create) and join the RestoNet peer group
        try {
            joinRestoNet(  );
        } catch (Exception e) {
            System.out.println("Can’t join or create RestoNet");
            System.exit(1);
        }

        // Wait for HungryPeers
        System.out.println("Waiting for HungryPeers");
    }
}
while (true) {
    // In later examples, HungryPeer requests are
    // processed here
    // For now, we’ll just block indefinitely
    synchronized(this) {
        try { wait(  ); } catch (InterruptedException ie) {} } 
    }
}

// Discover (or create) and join the RestoNet peergroup
private void joinRestoNet(  ) throws Exception {
    int count = 3;    // Maximum number of attempts to
discover
    System.out.println(
        "Attempting to Discover the RestoNet PeerGroup");

    // Get the discovery service from the NetPeergroup
    DiscoveryService hdisco = netpg.getDiscoveryService(  );
    Enumeration ae = null;    // Holds the discovered peers

    // Loop until we discover the RestoNet or
    // until we’ve exhausted the desired number of
    attempts
    while (count-- > 0) {
        try {
            // Search first in the peer’s local cache to
            find
            // the RestoNet peergroup advertisement
            ae = hdisco.getLocalAdvertisements(
                DiscoveryService.GROUP, "Name",
                "RestoNet");

            // If we found the RestoNet advertisement, we
            are done
            if ((ae != null) &amp; ae.hasMoreElements(  ))
                break;

            // If we did not find it, we send a discovery
            request
            hdisco.getRemoteAdvertisements(null,
                DiscoveryService.GROUP, "Name",
                "RestoNet", 1, null);

            // Sleep to allow time for peers to respond to
            the
            // discovery request
            try {
                Thread.sleep(timeout);
            } catch (InterruptedException ie) {} } 
        } catch (IOException e) {
            // Found nothing! Move on.
PeerGroupAdvertisement restoNetAdv = null;

// Check if we found the RestoNet advertisement. If we didn’t, then
// either we are the first peer to join or no other
RestoNet peers
// are up. In either case, we must create the RestoNet
peergroup.

if (ae == null || !ae.hasMoreElements()) {
    System.out.println("Could not find the RestoNet peer
group; creating one");
    try {
        // Create a new, all-purpose peergroup
        ModuleImplAdvertisement implAdv =
            netpg.getAllPurposePeerGroupImplAdvertisement();

        restoNet = netpg.newGroup(
            mkGroupID(),       // Assign new
group ID
            implAdv,           // The
implementation
            "RestoNet",        // Name of
peergroup
            "RestoNet, Inc."); // Description

        System.out.println("Found the RestoNet Peer
group advertisement; " +

        restoNetAdv = netpg.getPeerGroupAdvertisement();
    }
    ) catch (Exception e) {
        System.out.println("Error in creating RestoNet Peergroup");
        throw e;
    } else {
        // The RestoNet advertisement was found in the
        cache; // this means we can join the existing RestoNet
        peergroup
        try {
            restoNetAdv = (PeerGroupAdvertisement)
                ae.nextElement();
            restoNet = netpg.newGroup(restoNetAdv);
            System.out.println("Found the RestoNet Peergroup
tagvertisement; " +

            restoNetAdv = (PeerGroupAdvertisement)
As with all JXTA applications, the first thing the RestoPeer must do is discover and join the NetPeerGroup. Once it has joined this peer group, it can use the services of this peer group. In this case, it uses the discovery service (hdisco) of the NetPeerGroup object to look for the RestoNet peer group advertisement. We follow the standard steps outlined above. Check the local cache. If nothing is found, use the PDP to find the advertisement and sleep for a while to give the advertisement a chance to be discovered.

We try to discover the advertisement three times. Due to the unpredictable nature of the network, there is no way to tell how long it will take to find an advertisement. The number of attempts depends on your network environment: you may have to adjust the number of tries if you are running through multiple hops, or you may need to sleep longer to allow the message to come back. In general, these decisions must be made on an application-specific basis.

Note that this example uses JXTA's synchronous discovery API. The JXTA API also provides an asynchronous mechanism that an application can use to be notified when a resource is discovered (we show this API in Chapter 5). The synchronous discovery API provides a fine-grained control over the discovery process, while the asynchronous API is easier to manage.

Since we are looking for a peer group advertisement, the Discovery.GROUP type is used in the getLocalAdvertisements( ) method. The Name tag field of the peer group advertisement is used to search for the peer group. We are looking for a peer group advertisement in which the Name tag has a value equal to "RestoNet." Any peer group advertisements held in the local cache and with a matching type and attribute/value pair will be returned by this method. If such an advertisement is found, we are done, and we exit from the discovery loop.

If we did not find the advertisement in the local cache, a discovery request is sent within the NetPeerGroup via the getRemoteAdvertisements( ) method to find the RestoNet advertisement. Since the peer ID is null, the discovery request is propagated to all peers in the
current subnet and to the known rendezvous peers for the NetPeerGroup. The threshold argument for the request is set to 1, since there is only one RestoNet peergroup advertisement.

After the discovery request is sent, we sleep for a while before checking the cache to see if some responses were received. When a response is received, the advertisement is automatically stored in the local cache, but we need to allow some time for that process to happen. Note that a new remote discovery request is sent during each iteration. We must send multiple requests since the JXTA network does not guarantee that a discovery request will be processed by any peers. A rendezvous peer may be overloaded and discard the request. A new rendezvous may be available at the time of the request; as the JXTA network grows, more peers may be willing to respond.

If after three iterations the RestoNet advertisement was not found, then we continue on and create the RestoNet peergroup. We may be the first peer trying to join the RestoNet group, or all the other peers that know about the RestoNet group may be down or unreachable. We don’t know how many other peers are out there, or the state of any previous RestoNet peers; we simply know that, at this point in time, we cannot find the RestoNet peergroup.

4.3.1 Creating the Peergroup

If the RestoNet advertisement is not found, we must create a new RestoNet peergroup. We use the `getAllPurposePeerGroupImplAdvertisement()` method of the `PeerGroup` class to create a peergroup advertisement. This creates a generic advertisement; it automatically sets all of the attributes of the peergroup advertisement based on the NetPeerGroup (or, more generally, based on whichever peergroup is being used as the parent of the new peergroup).

The peergroup advertisement can then be passed to the `newGroup()` method of the NetPeerGroup. This method will create the new peergroup and send information about the new peergroup to the JXTA network (so that other peers can discover the newly created group). This is called publishing the advertisement; the `newGroup()` method publishes automatically, but you generally must explicitly publish other advertisements.

If we discovered an existing RestoNet advertisement, then the procedure is slightly altered: the RestoNet advertisement itself is used to instantiate the peergroup (though it still calls the `newGroup()` method of the NetPeerGroup). In this case, the `newGroup()` method will join the existing RestoNet peergroup; it will not need to publish the RestoNet advertisement as that advertisement has already been published.

4.3.1.1 The peergroup ID

It is important to point out that if peers cannot find the RestoNet peergroup, they need to create the same RestoNet peergroup; we don’t want the situation in which different peers create different versions of the RestoNet peergroup. This is done by ensuring that the peergroup ID generated by each peer is the same.

In our examples, we do this by using the `makeGroupID()` method in conjunction with a predefined string from which the URL defining the peergroup ID is generated. The string can be obtained via the `mkpgrp` command of the JXTA Shell by following these commands:

```
JXTA> mkpgrp dummygroup
JXTA> groups
group0: name = dummygroup
JXTA> cat group0
<?xml version="1.0">
<!DOCTYPE jxta:PGA>
<jxta:PGA xmlns:jxta="http://jxta.org">
  <GID>
    urn:jxta:uuid-4d6172676572696e204272756e6f202002
  </GID>
</jxta:PGA>
```
4.4 The Application Implementation

That completes our look at the RestoPeer implementation; we now have a service that can discover (and create, if necessary) the RestoNet peergroup. Now we'll turn our attention to the HungryPeer class.

The HungryPeer class represents an individual that is looking to eat at a restaurant. The implementation in this chapter illustrates how a peer can search for and join a peergroup: the HungryPeer class discovers and joins the RestoNet peergroup created by the previous RestoPeer class:

```java
import java.io.*;
import java.util.Enumeration;
import java.util.Vector;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.exception.PeerGroupException;
import net.jxta.discovery.DiscoveryService;
import net.jxta.pipe.PipeService;
import net.jxta.protocol.PeerGroupAdvertisement;

// HungryPeer tries to find a restaurant to get a good meal.
// It finds all
// RestoPeers in the RestoNet peergroup and (in later
// examples) requests
// bids from them.

public class HungryPeer {
    private PeerGroup netpg = null;  // NetPeerGroup
    private PeerGroup restoNet = null;  // Resto peer group
    private int timeout = 3000;       // Time-out; can be adjusted

    private DiscoveryService disco;   // Discovery service
    private PipeService pipes;        // Pipe service

    private Vector restaurantAdvs = new Vector(  );

    public static void main(String args[]) {
        HungryPeer myapp = new HungryPeer(  );
        myapp.startJxta(  );
        System.exit(0);
    }
```
private void startJxta() {
    try {
        // Discover (or create) and join
        // the default JXTA NetPeerGroup
        netpg = PeerGroupFactory.newNetPeerGroup();
    } catch (PeerGroupException e) {
        // Couldn’t initialize; can’t continue
        System.out.println("Fatal error : creating the net PeerGroup");
        System.exit(1);
    }

    // Discover and join the RestoNet peergroup
    // HungryPeers never create the RestoNet peergroup
    try {
        if (!joinRestoNet()) {
            System.out.println("Sorry could not find the RestoNet Peergroup");
            System.exit(2);
        }
    } catch (Exception e) {
        System.out.println("Can’t join RestoNet group");
        System.exit(1);
    }
}

private boolean joinRestoNet() {
    int count = 3; // Maximum number of attempts to discover
    System.out.println("Attempting to discover the RestoNet Peergroup");

    // Get the discovery service handle from the NetPeerGroup
    DiscoveryService hdisco = netpg.getDiscoveryService();

    // All discovered RestoNet peers
    Enumeration ae = null;

    // Loop until we find the RestoNet peergroup advertisement
    // or we’ve exhausted the desired number of attempts
    while (count-- > 0) {
        try {
            // Check if we have the advertisement in the local
// peer cache
ae = hdisco.getLocalAdvertisements(
    DiscoveryService.GROUP, "Name",
    "RestoNet");

// If we found the RestoNet advertisement, we are done
if ((ae != null) && ae.hasMoreElements( ) )
    break;

// The RestoNet advertisement is not in the local cache. Send a discovery request to search for it.
hdisco.getRemoteAdvertisements(null,
    DiscoveryService.GROUP, "Name",
    "RestoNet", 1, null);

// Wait to give peers a chance to respond
try {
    Thread.sleep(timeout);
} catch (InterruptedException ie) {} 
} catch (IOException e) {
    // Found nothing! Move on.
}

// Check if we found the RestoNet advertisement
if (ae == null || !ae.hasMoreElements( ) ) {
    return false;
}

System.out.println("Found the RestoNet PeerGroup Advertisement");
// Get the advertisement
PeerGroupAdvertisement adv =
    (PeerGroupAdvertisement) ae.nextElement( );

try {
    // Call the peergroup factory to instantiate a new peergroup instance
    restoNet = netpg.newGroup(adv);
} catch (Exception e) {
    System.out.println("Could not create RestoPeerGroup");
    return false;
}

try {
    // Get the discovery and pipe services for the RestoNet
    // peergroup (unused in this example)
    disco = restoNet.getDiscoveryService( );
    pipes = restoNet.getPipeService( );
} catch (Exception e) {
```java
System.out.println("Error getting services from RestoNet");
    throw e;
}
System.out.println("The HungryPeer joined the restoNet PeerGroup");
    return true;
}
```

Just as the RestoPeer service did, the HungryPeer starts by discovering and joining the NetPeerGroup. The basic outline of the `joinRestoNet()` method of the HungryPeer is very similar to the one in RestoPeer, except that this `joinRestoNet()` method does not create the RestoNet if one is not found. Instead, if this application does not find an existing RestoNet peergroup, it exits.

Not being able to find the RestoNet advertisement does not necessarily mean that no RestoNet peers are running in the JXTA network; it means only that we were not able to reach any RestoNet peers. This could be because we either did not wait long enough for an answer to come back, or we could not find a route to reach a running RestoNet peer. Trying for a total of 30 seconds is appropriate for this example, but you must make this decision in each application based on the demands of the application.

Note that there's no particular reason why the HungryPeer couldn't create the RestoNet peergroup if it didn't find one; it could wait for a RestoPeer to come along and join that peergroup. This is one reason why the RestoPeer searched for an existing peergroup before creating one.

### 4.5 Running the Example

To run both the RestoPeer and HungryPeer applications you must create two separate directories. Place the `HungryPeer.java` file in one directory, and the `RestoPeer.java` file in another directory. Each source file is compiled as we discussed in Chapter 3. The applications must exist in separate directories so that their configuration information will remain separate.

Let's compile and start the HungryPeer first and see what happens:

```bash
piccolo% cd ch04/HungryPeer
piccolo% javac HungryPeer.java
piccolo% java HungryPeer
```

Security initialization in progress.
This will take 10 or more seconds ...

Attempting to discover the RestoNet Peergroup
Sorry could not find the RestoNet Peergroup

When you start the application, you're first presented with the configurator. After filling it out, you'll see the output listed above. The HungryPeer application attempts to discover the RestoNet peergroup and aborts after 3 tries (a total of 30 seconds). This is what we'd expect, since the RestoPeer application has not yet created the RestoNet peergroup.

Oddly enough, when you run this example you may find instead that the HungryPeer has joined the RestoNet peergroup. There is no black magic here; this just means that someone else has already run the RestoPeer service on your JXTA network. Similarly, once the HungryPeer has found the RestoNet peergroup, it caches this information; if you rerun the HungryPeer at a later point, it will report that it has found the RestoNet peergroup even if no instances of the RestoPeer service are presently running on your network.

Let's now compile and run the RestoPeer application:

```bash
piccolo% cd ch04/RestoPeer
piccolo% javac RestoPeer.java
piccolo% java RestoPeer
```

Attempting to Discover the RestoNet PeerGroup
Could not find the RestoNet peergroup; creating one
RestoNet Restaurant (Chez JXTA) is on-line
Waiting for HungryPeers

Again, the application first runs the configurator. Remember that if you are running both peers on
the same host, you'll need to use the Advanced panel of the configurator to change the ports of one
of the peers (e.g., change the TCP port to 9703 and the HTTP port to 9702). After you complete
the configurator, the program continues and prints the output listed above.
RestoPeer attempts first to find the RestoNet peergroup. After approximately nine seconds (three
tries at three seconds each to discover the advertisement), RestoNet stops searching.
RestoPeer then creates a new RestoNet peergroup advertisement and joins the RestoNet peergroup
as the first peer. As before, if another RestoPeer has run on your network, this service will
discover and join the existing RestoNet peergroup.

4.5.1 Peer Cache Manager

After running the HungryPeer or RestoPeer, you may have noticed that a new directory (.jxta) was
created under the current directory, which contains several other directories. The cm directory is
the persistent cache for the platform; it contains advertisements that have been cached on the local
peer. Here's what the contents of .jxta might look like:

```
cm:
total 0
drwxr-xr-x  1 sdo    unknown         0 Jan  5 16:33 info-
jxta-NetGroup
  drwxr-xr-x  1 sdo    unknown         0 Jan  5 16:33 info-
jxta-WorldGroup
  drwxr-xr-x  1 sdo    unknown         0 Jan  5 16:34 info-
    uuid-2FD9ECF01D3D4B5CB4A9B8A67B4CC51F02
  drwxr-xr-x  1 sdo    unknown         0 Jan  5 16:33 jxta-
    NetGroup
  drwxr-xr-x  1 sdo    unknown         0 Jan  5 16:33 jxta-
    WorldGroup
  drwxr-xr-x  1 sdo    unknown         0 Jan  5 16:34 uuid-
    2FD9ECF01D3D4B5CB4A9B8A67B4CC51F02
```

The cm directory contains an info- and a jxta- directory that correspond to each peergroup; in this
case the uuid-2FD9ECF01D3D4B5CB4A9B8A67B4CC51F02 suffix corresponds to the RestoNet
peergroup. You may want to look at the cm directory after running each example later in the book
to see how the local cache of RestoPeers and HungryPeers is populated.
Advertisements are published with a time-to-live value that controls their lifetime; this value is
specified by whoever first publishes the advertisement. When the advertisement expires, JXTA
removes it from the cache.

4.5.1.1 Troubleshooting the cache

The presence of the cache has its advantages and disadvantages. It can greatly speed up discovery,
since it contains all the information the application has previously discovered. On the other hand,
there's no guarantee that the advertisements in the cache are still valid; they may point to obsolete
resources. This is integral to the notion of a P2P network; peers and their resources are known to
be unreliable or transient.
There's a balancing act going on here. You want to use the cache because it helps the performance
of your application. At the same time, you don't want to keep the cache forever, because over time
many of the advertisements that it contains will become invalid. To help prevent many of the
problems that can occur with stale advertisements, JXTA ages its advertisements. Still, you may
periodically need to reset the cache, particularly when you're testing new applications or have not
run the application for some time. To reset the cache, delete the entire cm directory before
restarting the platform.
Conversely, you can manually add advertisements to your cache in place of discovery; to do so, simply copy the file from an existing cache (one created by another application).

4.5.2 Rerunning the Applications

Let’s run RestoPeer again and see what happens. The RestoPeer now prints out these lines:

```
Attempting to Discover the RestoNet PeerGroup
Found the RestoNet Peergroup advertisement; joined existing group
RestoNet Restaurant (Chez JXTA) is on-line
Waiting for HungryPeers
```

RestoPeer tries to discover the RestoNet advertisement. It finds the advertisement right away and joins the RestoNet peergroup. The discovery is much faster since the RestoNet advertisement is now cached by the peer, and the RestoPeer only needs to perform a local discovery and join the peergroup.

With the RestoPeer still running, let’s again run the HungryPeer. We get this output:

```
Attempting to discover the RestoNet Peergroup
Found the RestoNet PeerGroup Advertisement
The HungryPeer joined the restoNet PeerGroup
```

The HungryPeer tries to discover the RestoNet advertisement. After a few seconds, the "Found the RestoNet" message prints out, indicating that the peer found the RestoNet advertisement. HungryPeer uses the advertisement to instantiate and join the RestoNet peergroup.

Now when we run HungryPeer again, it prints out this output:

```
Attempting to discover the RestoNet Peergroup
Found the RestoNet PeerGroup Advertisement
The HungryPeer joined the restoNet PeerGroup
```

The difference is that now the "Found the RestoNet" message appears right away. Because the advertisement was cached, no discovery request needed to be sent. The HungryPeer instantiates and joins the RestoNet peergroup as before. This illustrates the capability of the HungryPeer to learn information and make discovery work faster the second time around.

4.6 Key Benefits of Discovery

Before we leave the topic of discovery, let’s see how our example fits with two key benefits of JXTA’s model of service and advertisement discovery.

4.6.1 Performance Improves as the System Ages

The ability to improve performance as the system ages is a fundamental feature of a JXTA network. As a peer discovers new resources, it becomes more capable and efficient. This helps the peer, of course, but it benefits other peers on the network as well.

The first time a peer attempts to discover another peer, it will take some time. The second time it will take less time, because the peer has cached the discoveries it has previously made. That helps the peer.

Remember, though, that advertisements get cached on all intermediary peers that they pass through. Therefore, advertisements in a JXTA network tend to move towards the end user’s peer; a peer does not always have to go back to a central location or to the publisher of the advertisement to obtain an advertisement. So the first HungryPeer to run on a local network may take some time to discover the RestoNet peergroup, particularly if the only RestoPeer on the network was just started many network hops away. The second HungryPeer will find the RestoNet more quickly; it has to find only the first HungryPeer or the RestoPeer, one of which is bound to be closer than the other. As the peergroup becomes more popular, the probability that a nearby peer has already discovered the peergroup increases, making the discovery process faster for new peers.

The JXTA network also provides a new kind of fault-tolerant environment that allows information to be replicated as it is used. An advertisement may be cached locally, it may be known by a
nearby peer or by a rendezvous peer, or a remote peer may have it. There is no unique path to obtain an advertisement. As advertisements are propagated and cached within the network, JXTA adapts to retrieve the advertisement from the most efficient location. In many cases, this location is the closest to the peer.

As a final example, consider what happens when you stop the RestoPeer and restart the HungryPeer. You get the same output we're used to seeing:

```
Attempting to discover the RestoNet Peergroup
Found the RestoNet PeerGroup Advertisement
The HungryPeer joined the restoNet PeerGroup
```

The HungryPeer finds the RestoNet advertisement in its cache and joins the RestoNet peergroup. JXTA removed the need for a centralized infrastructure to support the RestoNet peergroup. Any peer (RestoPeer or HungryPeer) that previously joined the RestoNet group has enough information to join the group again. In our case, we just have a hungry peer in the peergroup and no food to feed him! This is okay: when a RestoPeer joins the peergroup, it may try to look for our hungry peer.

### 4.6.2 Reliability from an Unreliable and Unpredictable Network

Programming in a P2P network such as JXTA is a little more involved than in a traditional, reliable network with a higher guarantee of predictability. However, by assuming unreliability in a network from the beginning, JXTA allows us to build more reliable applications in the long run. Many traditional network programming models provide the illusion of reliability by trying to mask network errors. In many situations, this leads to unpredictable and intractable behaviors when an application is deployed in a real-world network. The programming model of JXTA, while giving the appearance of more complexity, may lead to more robust and predictable network applications.

This is a fundamental programming shift that JXTA introduces. We strongly believe that the JXTA programming model will lead to more robust distributed applications by forcing applications to leverage all of their surrounding environments and dynamically bind themselves to interchangeable network resources. Network failures are treated not as special cases, but as normal operations. Network resources should be interchangeable, and peer location should be irrelevant.

Consider the scenario of a man driving at night, searching for a restaurant. The man may decide to search until his stomach tells him, "Enough is enough. Give me some food!" So instead of continuing to search for a restaurant, the man may stop at a grocery store he is just passing by, even though a restaurant on the next block happens to be open. If the man had continued driving one more block, he would have discovered the restaurant.

This is exactly what happens in real life. Different resources are available, new resources may be just around the corner or just about to open, and we make decisions based on our knowledge available at the time.

### 4.7 Summary

In this chapter, we looked at the Java bindings that allow peers to discover service advertisements. All service advertisements are discovered in essentially the same way. We've also shown how a peer can join a peergroup once a peergroup advertisement is found. In addition, we've shown how to construct and publish service advertisements when they cannot be found.

The discovery of service advertisements (and, specifically, peergroup advertisements) is a key feature of the JXTA platform. JXTA does not rely on a central registry of services or any other piece of infrastructure; all that two peers need in order to communicate with each other is a network. As more peers join the network and discover common advertisements, the JXTA network becomes more robust.

At this point our peers have found each other, but they have no additional communication. In the next chapter, we'll see how peers in a peergroup can communicate via pipes.
Chapter 5. The Pipe API

In this chapter, we'll expand on our restaurant auction example from Chapter 4 to show how the RestoPeer and HungryPeer communicate with each other once they have each discovered and joined the RestoNet peergroup. From a functional point of view, this communication is the basis for what the distributed application does; communication between the peers enables the RestoPeer to offer services and the Hungry Peer to place bids on those services.

From a programming point of view, this communication is possible due to the Java API binding that JXTA provides for pipes. Pipes are the fundamental construct of JXTA that allow peers to communicate with each other; a peer that wants to offer a service creates and advertises a pipe to the JXTA peergroup. A peer that wants to use a service discovers pipe advertisements in a peergroup and makes a connection to the service via the pipe.

Once again, we'll show only the Java language binding for JXTA pipes in this chapter. Because JXTA defines the protocol by which pipes are advertised and used, any JXTA peer can communicate with any other JXTA peer over the pipe; the two peers do not need to be programmed in the same language.

5.1 Creating Pipes

We'll illustrate the use of pipes by modifying our previous auctioning example; we will add a set of pipes to allow HungryPeers and RestoPeers to communicate within the RestoNet peergroup. Since pipes are unidirectional, we need two pipes for a HungryPeer and RestoPeer to exchange messages: one for the HungryPeer to send a request to a RestoPeer, and one for the RestoPeer to respond to the HungryPeer.

To support this, the RestoPeer class must be modified to create and publish its pipe advertisement. The RestoPeer pipe advertisement is used by HungryPeers to discover the associated RestoPeer and create an output pipe endpoint through which it connects to the RestoPeer pipe. When the pipe connection is established, HungryPeers use the RestoPeer pipe to send auction requests to the RestoPeer.

As an example, HungryPeers send auction requests for French fries to RestoPeers. A HungryPeer may request three sizes of French fries (small, medium, and large). RestoPeers respond to fries auctions by sending a price offer for the size requested. Each RestoPeer will respond with its own price. The intent is for the HungryPeer to select the offer with the lower price.

As part of their responses, RestoPeers also send a special offer for French fries available at the RestoPeer. Looking at special offers and standard offers from all RestoPeers that send a response to an auction request, our HungryPeer will decide which RestoPeer to select.

Here's the code for our modified RestoPeer class:

```java
import java.io.*;
import java.net.*;
import java.util.*;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.exception.PeerGroupException;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.Advertisement;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.discovery.DiscoveryService;
import net.jxta.id.IDFactory;
import net.jxta.pipe.PipeService;
```
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.PipeID;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;
import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.endpoint.Message;
import net.jxta.id.IDFactory;

// RestoPeer represents a restaurant that receives auction requests for
// French fries from HungryPeers. It offers three sizes of French fries
// (small, medium, large). Each restaurant assigns a different price for
// each size and offers a special. Each restaurant is uniquely identified by
// its brand name.

public class RestoPeer {
    private PeerGroup netpg = null; // The NetPeerGroup
    private PeerGroup restoNet = null; // The RestoNet peer group

    private String brand = "Chez JXTA"; // Brand of this restaurant
    private String specials = "large ($3.00)"; // Current special

    // Services within the RestoNet peer group
    private DiscoveryService disco = null; // Discovery service
    private PipeService pipes = null; // Pipe service
    private PipeAdvertisement myAdv = null; // My pipe advertisement
    private InputPipe pipeIn = null; // Input pipe that we
                                     // listen to for requests

    private int time-out = 3000; // Discovery wait time-out
    private int rtime-out = 8000; // Resolver pipe time-out

    static String groupURL =
        "jxta:uuid-4d6172676572696e204272756e6f202002";

    public static void main(String args[]) {
        RestoPeer myapp = new RestoPeer();
        myapp.startJxta();
        System.exit(0);
    }
}
private void startJxta() {
    try {
        // Discover and join (or start) the default
        // Peergroup
        PeerGroupFactory.newNetPeerGroup();
    } catch (PeerGroupException e) {
        // Couldn’t initialize; can’t continue
        System.out.println("Fatal error: creating the NetPeerGroup");
        System.exit(1);
    }

    // Discover (or create) and join the RestoNet Peergroup
    try {
    }
    catch (Exception e) {
        System.out.println("Can’t join or create RestoNet");
        System.exit(1);
    }

    // Discover (or create) and publish a RestoPeer pipe to receive
    // auction request for fries from HungryPeers
    if (!createRestoPipe()) {
        System.out.println("Aborting due to failure to create RestoPeer pipe");
        System.exit(1);
    }

    // Start the RestoPeer server loop to respond to
    // HungryPeer’s
    // fries requests
    handleFriesRequest();
}

private void handleFriesRequest() {
    // See later example in this chapter
}

private boolean createRestoPipe() {
    // Create the RestoPeer pipe associated with this
    // RestoPeer. First,
    // discover if a pipe advertisement exists. If it doesn’t,
    // create and
    // publish it.
    private boolean createRestoPipe() {
int count = 3; // Discovery retry count
Enumeration ae = null; // Discovery response

try {
    System.out.println("Attempting to Discover the Restaurant
RestoPipe");

    // Check if we have already published ourselves
    while (count-- > 0) {
        try {
            // First, check locally if the advertisement is cached
            ae = disco.getLocalAdvertisements(
                DiscoveryService.ADV, "name",
                "RestoNet:RestoPipe:" + brand);

            // If we found our pipe advertisement, we are done
            if (ae != null && ae.hasMoreElements( ) )
                break;

            // We did not find the advertisement locally;
            // send a remote request
disco.getRemoteAdvertisements(null, 
                DiscoveryService.ADV, "name",
                "RestoNet:RestoPipe:" + brand, 1,
                null);

            // Sleep to allow time for peers to respond to the
            // discovery request
            try {
                Thread.sleep(time-out);
            } catch (InterruptedException e) {} 
        } catch (IOException e) {
            // Found nothing! Move on.
        }
    }

    if (ae == null || !ae.hasMoreElements( ) ) {
        // We did not find the pipe advertisement, so create one
        System.out.println("Could not find the Restaurant Pipe Advertisement");

        // Create a pipe advertisement for our RestoPeer
        myAdv = (PipeAdvertisement)
            AdvertisementFactory.newAdvertisement(
                PipeAdvertisement.getAdvertisementType( ) );
    }
// Assign a unique ID to the pipe
IDFactory.newPipeID(restoNet.getPeerGroupID());

// The symbolic name of the pipe is built from the brand
// name of RestoPeer; each RestoPeer must therefore have a
// unique name
myAdv.setName("RestoNet:RestoPipe:" + brand);

// Set the type of the pipe to be unidirectional
myAdv.setType(PipeService.UnicastType);

// We have the advertisement; publish it into our local
// cache and to the RestoNet peergroup. We use the default
// lifetime and expiration time for remote publishing.
disco.publish(myAdv, DiscoveryService.ADV,
PeerGroup.DEFAULT_LIFETIME,
PeerGroup.DEFAULT_EXPIRATION);
disco.remotePublish(myAdv,
DiscoveryService.ADV,
PeerGroup.DEFAULT_EXPIRATION);

System.out.println("Created the Restaurant Pipe Advertisement");
} else {
    // We found an existing pipe advertisement
    myAdv = (PipeAdvertisement) ae.nextElement();

    System.out.println("Found Restaurant Pipe Advertisement");
}

// Create my input pipe to listen for HungryPeer’s requests
pipeIn = pipes.createInputPipe(myAdv);
} catch (Exception e) {
    System.out.println("Could not initialize the Restaurant pipe");
    return false;
}

return true;

Most of this code is identical to our previous implementation of the RestoPeer; in particular, the startJxta() method uses the joinRestoNet() method to discover (or create) and join the RestoNet peergroup. After finding the peergroup, it does something new: it creates the
pipe with the `createRestoPipe()` method. After the pipe is created, it processes messages on the pipe by calling the `handleFriesRequest()` method (which we discuss in a later example).

So the new code here is all within the `createRestoPipe()` method. Logically, we create the pipe the same way we created a peergroup in Chapter 4:

- We search for the existing advertisement in our local cache using the `getLocalAdvertisements()` method. Note that we use `DiscoveryService.ADV` as the type of advertisement (since we're not looking for a peer or peergroup advertisement).
- If it's not there, we call the `getRemoteAdvertisements()` method to attempt to discover the advertisement. We wait three seconds to allow the discovery to continue and again look in the local cache to see if we found it. This process is repeated three times.

Note that the discovery service used in these two steps is now the discovery service from the RestoNet peergroup (rather than the discovery service from the NetPeerGroup that we used in the `joinRestoNet()` method). As we have mentioned, each peergroup has its own instance of the discovery service (and all other services) for scoping and security reasons. The pipes that we use for the auctioning service are published and discovered only within the context of the RestoNet peergroup.

- If we still haven't found the advertisement, we must create one. We use the `AdvertisementFactory` class to construct the advertisement and then set the appropriate information in the advertisement, including the name of the pipe ("RestoNet:RestoPipe:Chez JXTA"), its unique ID, and the type of the pipe (`PipeService.UnicastType`).
- Once we have the pipe advertisement (because we discovered it or because we created it), we send it to the pipe service in order to instantiate the input pipe. This is the pipe over which the RestoPeer will listen for requests. Later, the RestoPeer will need to create an output pipe over which it can send responses to specific HungryPeers.

Let's look at a few of the key classes and methods that we used to create the pipe advertisement.

### 5.1.1 The `PipeAdvertisement` Class

When we created the pipe advertisement, we specified that the factory should create an instance of the `PipeAdvertisement` class (`net.jxta.protocol.PipeAdvertisement`). A pipe advertisement contains three attributes: the name of the pipe, the type of the pipe, and the unique ID of the pipe. We must set each of these attributes before the advertisement is published; this is done by using the `setPipeID()`, `setType()`, and `setName()` methods of the `PipeAdvertisement` class.

### 5.1.2 The `PipeID` Class

When we create the pipe advertisement, we must give it a new ID that is unique across the JXTA network. We do this with the `PipeID` class (`net.jxta.pipe.PipeID`), which guarantees that the IDs it generates are unique.

Pipe IDs are created by the `IDFactory` class (`net.jxta.id.IDFactory`). This class has static methods that create all types of JXTA IDs; for pipe IDs, the `newPipeID()` method takes as an argument a `PeerGroupID` that identifies the peergroup this pipe will belong to. The internal representation of a pipe ID is a combination of a peergroup ID and a unique pipe ID. From a pipe ID, you can use the `getPeerGroupID()` method to extract the peergroup ID.

### 5.1.3 Types of Pipes
In the pipe advertisement, we had to specify the type of pipe we wanted. The default core pipe service provided by JXTA supports three kinds of pipes:

**UnicastType**

Unicast means that the pipe is a point-to-point pipe: a message sent on the pipe will be directed to a single network address (as opposed to a broadcast or multicast address). Unicast pipes are unreliable, and unsecure. "Unreliable" means that there are no assurances that messages sent on the pipe will make it to their destination, and messages sent on the pipe may be received out of order. "Unsecure" means that the data transmitted over the pipe is unprotected: intruders on the network can view, and in some cases modify, the data as it passes through the pipe.

**PropagateType**

Propagate pipes are 1-n pipes: a message written on a propagate pipe may be read by multiple recipients. Propagate pipes are implemented as broadcast or multicast sockets on IP networks. Data sent on a propagate pipe is unsecure.

**UnicastSecureType**

Unicast secure pipes are unreliable and secure. Secure pipes have two properties:

- Parties that connect to a secure pipe are authenticated; a peer is assured of the identity of the peer on the other end of the pipe.
- Data that travels over the pipe is encrypted; it cannot be read or modified in transit.

These attributes are the minimum attributes required by JXTA implementation. In practice, implementations may use whatever underlying transport features are available. On an IP network, unicast pipes are generally implemented as TCP sockets, which means that messages that travel over the pipe are guaranteed to arrive at their destination in order. However, JXTA applications cannot count on this fact, and an application that uses a unicast pipe must be prepared for out-of-order messages. In fact, even on a TCP/IP network, JXTA messages may be relayed by one or more intermediary peers that provide NAT or firewall traversal; these peers may drop messages due to message queue overflow.

It's possible, of course, to use a unicast pipe and still secure its data: an application can use the APIs discussed in Chapter 7 to encrypt data that is sent and to identify parties as they connect to a pipe. A secure unicast pipe does this automatically, making the application-level programming easier. We'll discuss secure unicast pipes further in Chapter 7.

### 5.1.4 Inconsistent Advertisements

In our example, the name of the pipe is "RestoNet:RestoPipe:JXTA." It is important for this name to be unique so that other peers can associate this pipe with our RestoPeer. This code assumes that only one restaurant is named "Chez JXTA." If it is not the only one, a unique branch number (or other unique identifier) could be added.

In general, it is good practice to search for an advertisement before publishing it to ensure that someone else has not already published the advertisement. Republishing a pipe advertisement is not an issue as long as the binding defined by the pipe name and the pipe ID is unique. Remember that many people may be running the RestoPeer example on the JXTA network. You do not want to introduce two inconsistent advertisements (advertisements with the same name, but a different pipe ID); this is why you should search for the advertisement before publishing it. It's also why you should make the name of the pipe unique (i.e., before you run this code on a machine connected to a network, change the brand name). Otherwise, you'll end up creating a pipe that hears messages that are really destined for someone else's peer.

Conflicts between inconsistent advertisements will typically be resolved by the creation of the pipe or via an advertisement aging (time-to-live) mechanism that purges obsolete advertisements. If our HungryPeer finds more than one advertisement, it will keep trying advertisements until it can connect to the pipe.

To ensure the validity and origin of advertisements and to protect against fraudulent attacks, advertisements may be signed with a digital signature that uniquely identifies its publisher.
5.1.5 Publishing Advertisements

In our previous examples, advertisements were published automatically: the methods of the
peergroup class automatically created and published the peergroup advertisements that allowed
HungryPeers to find the RestoNet peergroup.
Advertisements other than peergroup advertisements must be published explicitly if you want
them to be discovered by other peers. There are times when you don’t want to publish the
advertisement. But we do want HungryPeers to be able to discover RestoPeers, so the RestoPeer
must publish its pipe advertisement. It does this by making two calls to the discovery service.
The first call is to the publish( ) method, which places the advertisement in the local cache.
This enables RestoPeer to find the same pipe advertisement the next time it is started. The call to
the remotePublish( ) method makes the advertisement available to the peergroup (and
therefore, to other peers who have joined the peergroup).
Peers who discover this advertisement will keep the advertisement in their local cache (as will the
peer that published the advertisement). As we discussed in Chapter 4, it is sometimes necessary to
clear the cache of stale advertisements. Advertisements are held in the cache for a duration
specified when the advertisement is published; in this case, we’ve used PeerGroup.DEFAULT_EXPIRATION
(which happens to be two weeks) as the duration. Whenever you publish an advertisement, you should determine an appropriate duration for that
advertisement; for a test program like this, a few hours might work well. For a server you expect
to be up continually, a few weeks or months might work well.
Note that when the advertisement is placed in the local cache, you can specify how long the
advertisement should live in the local cache (PeerGroup.DEFAULT_LIFETIME, or one year
in this case). Long values are reasonable for this, since the application is the only thing that will
reuse the advertisement.

5.1.6 The Pipe Service

Once the RestoPeer pipe advertisement is created and published, we must create an input pipe
endpoint to listen for messages. To do this, we use the Pipe service, which implements the Pipe
interface (net.jxta.pipe.Pipe). This is a core service of peergroups; we retrieved it from
the RestoNet peergroup (using the getPipe( ) method) in the joinRestoNet( ) method.
The Pipe interface allows you to create input and output pipe endpoints and create messages to
be sent over those pipes. Input endpoints are created with the createInputPipe( ) method,
which takes as its argument a pipe advertisement that corresponds to the pipe that this pipe
endpoint will be connected to; we save the input pipe for the RestoPeer in the pipeIn variable.
Output endpoints are created with the createOutputPipe( ) method, which we’ll see in
the next section.

5.2 Pipe Messages

The next step in the implementation of our RestoPeer is to implement its
handleFriesRequest( ) method. This method is responsible for reading requests from
HungryPeers and sending responses to them; it shows us how to pass messages over pipes.
Here’s how we implement this method:
// Method to handle fries auction requests from HungryPeers.
The method
// waits for HungryPeer requests’ pipe messages to arrive.
Incoming requests
// contain a pipe advertisement for responding to the
HungryPeer’s requester
// and a fries size. The method generates a bid offer for the
request, opens
// an output pipe to the HungryPeer requester, and sends the
response.
private void handleFriesRequest( ) {
    InputStream ip = null; // Input stream to read message
    PipeAdvertisement hungryPipe = null; // HungryPeer requester pipe
    StructuredDocument request = null; // Request document
    StructuredDocument bid = null; // Response document
    // Document mime types
    MimeMediaType mimeType = new MimeMediaType("text", "xml");
    Element el = null; // Element in document
    String name = null; // Name of the sender
    String size = null; // Fries size requested
    OutputPipe pipeOut = null; // Output pipe to respond to
    // HungryPeer requester
    System.out.println("RestoNet Restaurant (" + brand + ") waiting for HungryPeer requests");

    // Loop waiting for HungryPeer requests
    while (true) {
        Message msg = null; // Incoming pipe message
        try {
            // Block until a message arrives on the RestoPeer pipe
            msg = pipeIn.waitForMessage( );
            // If message is null, discard message
            if (msg == null) {
                if (Thread.interrupted( )) {
                    // We have been asked to stop
                    System.out.println("Abort: RestoPeer interrupted");
                    return;
                }
            }
            // We received a message; extract the request
            try {
                // Extract the HungryPipe pipe information
                // to reply to the sender
                ip = msg.getElement("HungryPeerPipe").getStream( );
                // Construct the associated pipe advertisement
                // via the AdvertisementFactory
                hungryPipe = (PipeAdvertisement) AdvertisementFactory.newAdvertisement(mimeType, ip);
                // Extract the sender name and fries size requested,
// building a StructuredDocument
ip = msg.getElement("Request").getStream();
request = StructuredDocumentFactory
    .newStructuredDocument(mimeType, ip);

// Extract the fields from the structured document
Enumeration enum = request.getChildren();

// Loop over all the elements of the document
while (enum.hasMoreElements()) {
    el = (Element) enum.nextElement();
    String attr = (String) el.getKey();
    String value = (String) el.getValue();

    // Extract the HungryPeer requester name
    if (attr.equals("Name")) {
        name = value;
        continue;
    }

    // Extract the fries size requested
    else if (attr.equals("Fries")) {
        size = value;
        continue;
    }
}

} catch (Exception e) {
    continue; // Broken content; silently discard
}

System.out.println("Received Request from
HungryPeer "
    + name
    + " for 
    + size
    + " Fries.");

// The auction request is valid. We can create the output pipe
// for sending the response bid to the HungryPeer requester.
try {
    System.out.println(
        "Attempting to create Output Pipe to
HungryPeer "
        + name);

    // Create an output pipe connection to the HungryPeer
    pipeOut = pipes.createOutputPipe(hungryPipe, rtime-out);

    // Check if we have a pipe
    if (pipeOut == null) {
        // Cannot connect to the pipe
System.out.println("Could not find HungryPeer pipe");
    continue;
}
} catch (Exception e) {
    // Pipe creation exception
    System.out.println("HungryPeer may not be listening anymore");
    continue;
}

// We have a pipe connection to the HungryPeer
// Now create the bid response document
try {
    // Construct the response document
    bid = StructuredDocumentFactory.
        newStructuredDocument(mimeType,
            "RestoNet:Bid");

    // Set the bid values (brand, price, special)
    // in the response document
    el = bid.createElement("Brand", brand);
    bid.appendChild(el);
    el = bid.createElement("Price",
        friesPrice(size));
    bid.appendChild(el);
    el = bid.createElement("Specials", specials);
    bid.appendChild(el);

    // Create a new pipe message
    msg = pipes.createMessage();

    // Push the bid offer in the message
    msg.addElement(msg.newMessageElement(
        "Bid",mimeType, bid.getStream(
    )
    ));

    // Send the message
    pipeOut.send(msg);

    // Close the output pipe connection
    pipeOut.close();
} catch (Exception ex) {
    System.out.println("Error sending bid offer to HungryPeer " + name);
    continue;
}

System.out.println("Sent Bid Offer to HungryPeer " + name + 
    ") Fries price = "+ friesPrice(size) + 
            ", special = " + specials);
} catch (Exception e) {
    System.out.println("Abort RestoPeer interrupted");
private String friesPrice(String size) {
    if (size.equals("small"))
        return "$1.50";
    if (size.equals("medium"))
        return "2.50";
    if (size.equals("large"))
        return "3.00";
    return "error";
}

// Determine the price of the French fries depending on the size

Let’s look at the new APIs used by this code.

5.2.1 The Message Interface

The handleFriesRequest( ) method listens for auction request messages on the RestoPeer input pipe and sends bid responses to HungryPeers. Messages are objects that implement the Message interface (jxta.net.endpoint.Message). JXTA messages are multi-part messages; each part has a unique name and can be constructed or extracted separately by the sender or the receiver.

The contents of a message are defined by the application; JXTA places no restrictions on what they may contain. In our example, auction messages received by RestoPeers are composed of two parts:

HungryPeerPipe part
This part contains a HungryPeer pipe advertisement; the RestoPeer should send its bid to this pipe.

Request part
The request part contains the French fries auction request document. The request is represented as an XML document, which is implemented programatically by the StructuredDocument class, as follows:

```xml
<RestoNet:Request>
  <Name> Name of the HungryPeer Requester </Name>
  <Fries> size requested </Fries>
</RestoNet:Request>
```

After extracting and processing the auction request, a RestoPeer responds with a bid message to the HungryPeer. The outgoing bid offer message is composed of one part: the bid part. This part contains a bid document. The bid is represented as this XML:

```xml
<RestoNet:Bid>
  <Brand> name of the RestoPeer offering the bid </Brand>
  <Price> Price bid for the size requested </Price>
  <Specials> Specials offers available at this RestoPeer </Specials>
</RestoNet:Bid>
```

There are two ways to obtain a message: by reading it and by creating it.

5.2.1.1 Reading messages

Messages may be read from an input pipe by using the waitForMessage( ) method or the poll( ) method:

```java
waitForMessage()
```
This method blocks until a message arrives; it returns a new message. It will return null if the input pipe is closed (e.g., if the HungryPeer exits before the message can be read).

`poll()`

This method is nonblocking and takes a time-out argument. If no message is available within the given time, it will return null; if the time-out is zero, then the `poll()` method waits indefinitely. Input pipes can also be registered with a `PipeMsgListener` object, which will then be called when input is pending on the input pipe.

Once a message has been read, you must retrieve the individual parts of the message. This is accomplished with the `getElement()` method of the `Message` interface. The `getElement()` method returns an input stream from which the corresponding part of the message can be read.

Data is read from the input stream directly; the format of the data will vary depending on the type of the message part. In our example, we know that the section named HungryPeerPipe contains a pipe advertisement, and we know that in JXTA, all advertisements are represented as XML documents. Therefore, we can pass the input stream and an appropriate MIME type to the `newAdvertisement()` method of the `AdvertisementFactory` class. This reads in the advertisement information and returns the pipe advertisement over which we can contact the HungryPeer. Similarly, when we retrieve the request document, we can pass the input stream to the structured document factory in order to read the XML document directly.

### 5.2.1.2 Creating messages

Messages are created by the `createMessage()` method of the pipe service. Once a message has been created, data must be inserted into it (we discuss this process in the next section).

### 5.2.2 The StructuredDocument Class

In Chapter 3, we introduced the `Document` class, which is used to manipulate documents. In this example, we use one of its subclasses, the `StructuredDocument` class. Structured documents are composed of a hierarchy of elements, very much like XML documents. The `StructuredDocument` class allows us to manipulate the content of several document types in an abstract way, without regard to the physical representation of the documents.

#### 5.2.2.1 Reading structured documents

When we retrieve the input stream of the "request" message section of the auction request, we use it to create a structured document with the static `newStructuredDocument()` method of the `StructuredDocumentFactory` class. The `newStructuredDocument()` method accepts two objects as arguments:

- `MimeMediaType` Since the auction request is represented as an XML document, we pass a `MimeMediaType` object constructed from the string "text/xml."
- `InputStream` The input stream points to a stream that contains the data that will be read to construct the structured document.

After we extract the auction request document, the parsing methods of the `StructuredDocument` class get the various fields in the request. The `getChildren()` method returns an enumeration of all the fields in the document; each field pair (key and value) is represented as an `Element` object. The `getKey()` and `getValue()` methods of the `Element` interface (net.jxta.document.Element) get the key and value of each field; in our example, we use them to extract the "Name" and "Fries" fields of the auction request.

#### 5.2.2.2 Creating structured documents
Later in our example, we will create a structured document that represents the bid response to the HungryPeer. The bid response document is created via the `newStructuredDocument( )` method of the `StructuredDocumentFactory`, which in this case creates an XML document (since the MIME type is "text/xml").

The `StructuredDocument` interface has methods to manipulate its elements. In our example, we use two of these:

```java
public Element createElement(Object key)
public Element createElement(Object key, Object value)
```

This method creates a new element in a document. An element is identifiable by a key and may optionally have a value. Each element also maintains a collection of references to other elements (its children). Elements can be composed into arbitrary hierarchical structures, forming complex data structures. Element instances are always associated with a structured document. A structured document is a specialized form of element with additional features that make it an appropriate root of a hierarchy of elements; in fact, it extends the `Element` interface.

In our example, we use the `createElement( )` method to create each element of the XML tree that represents the bid document: the brand, price, and specials tags.

```java
public void appendChild(Element element)
```

This method appends a new child element to an element. No special ordering is maintained between the children. Note that this method actually comes from the `Element` interface.

So, to create structured documents, you create the elements and then append them to the existing document object. If the structured document has a complex hierarchy, you must repeat this technique recursively.

### 5.2.3 Creating Output Pipes

Now that all of the pieces are in place, we can discuss how the RestoPeer can process the request and generate the bid. We use the HungryPeer pipe advertisement to create an output pipe endpoint over which to send the bid response to the requesting HungryPeer. We use the `createOutputPipe( )` method of the RestoNet pipe service to create the output pipe endpoint with which to talk to the HungryPeer. The `createOutputPipe( )` method accepts two objects as arguments:

- **A pipe advertisement object**
  - The pipe advertisement corresponding to the pipe on which we want to create an output endpoint.

- **A time-out object**
  - The time-out specifies how long the call should wait to resolve the pipe (i.e., to find the input pipe endpoint corresponding to our output endpoint). A time-out value of \(-1\) means that the `createOutputPipe( )` method will block until the endpoint is resolved; otherwise, a time-out value is specified in milliseconds.

In our example, we used a time-out of eight seconds; depending on the network configuration, you may have to adjust the time-out. If you have the RestoPeers and HungryPeers running on the same computer (or on the same subnet), you may have to reduce the time-out. If you are relying on an HTTP router for RestoPeers to talk to the HungryPeers, you should keep the time-out value. If the router you are using is very busy, you may have to increase the time-out.

The HungryPeer pipe advertisement that is received is used to create the output pipe endpoint to send the offer to the HungryPeer. If we cannot resolve the pipe endpoint, we discard the request and wait for the next request. The endpoint resolution may fail for a number of reasons: the HungryPeer may have disappeared, we may no longer be able to reach the HungryPeer due to the failure of an intermediary router, or the resolve time-out may be too small, in which case the RestoPeer could try to resolve the pipe with a larger time-out.

Once the output pipe has been created, we initialize the bid document (since there's no point in creating the document if we can't talk to the HungryPeer). The bid document is stored into a new part (named "Bid") of the message via the `addElement( )` method of the `Message`
interface. Like its complementary `getElement()` method, the `addElement()` method requires an input stream as an argument. Therefore, we must convert the bid document to an input stream, which we do via the `getStream()` method of the `StructuredDocument` class. Finally, the bid message is sent to the HungryPeer via the `send()` method of the `OutputPipe` class. The `send()` method is asynchronous; there is no guarantee that the message is received before it returns (or that the message will be received at all). At this point, the auction request is processed. We close the output pipe and wait for the next message.

We must point out one more thing about the output pipe used here: the HungryPeer sent the RestoPeer a pipe advertisement, not a physical endpoint that binds the HungryPeer to a physical network address in the JXTA network. As we have discussed, pipes provide a nonlocalized, virtual communication abstraction, and this example illustrates the full power of JXTA pipes. Since only a pipe advertisement is sent, when the output pipe is created, the RestoPeer dynamically binds the HungryPeer pipe from the latest network location of the HungryPeer. From the time the request was sent, the HungryPeer could have moved to a new physical network address; it may be using wireless roaming, a different device, and so on. The RestoPeer will still be able to find and connect to the HungryPeer because of the virtual nature of JXTA's pipe advertisements.

JXTA pipes introduce a fundamentally different way to establish communication between services, irrespective of the physical location of a network service. Abstracting the entire network topology and network location enables the deployment of simpler and more robust service interactions as services rebind or fail-over to different instances of a service in a completely transparent manner.

### 5.3 Pipe Discovery

In this section, we'll look at the implementation of the HungryPeer. One new technique is that the HungryPeer must construct the message that it sends to the RestoPeer. With the new technique, the HungryPeer discovers the published pipe advertisement of a RestoPeer; in our last example, the RestoPeer *received* the pipe advertisement in the HungryPeer's auction message. Here's the implementation of the HungryPeer that discovers the pipe:

```java
import java.io.*;
import java.util.Enumeration;
import java.util.Vector;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.exception.PeerGroupException;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.Advertisement;
import net.jxta.document.StructuredDocument;
import net.jxta.document.Element;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.discovery.DiscoveryService;
import net.jxta.pipe.PipeService;
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.PipeID;
import net.jxta.pipe.OutputPipe;
import net.jxta.endpoint.Message;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;
import net.jxta.id.IDFactory;
```
// The HungryPeer joins the RestoNet peergroup and searches for RestoPeers.  
// The HungryPeer then establishes a pipe connection to all the RestoPeers  
// that it discovered. It sends auction requests for French fries to  
// RestoPeers and then waits for auction bids from RestoPeers.

public class HungryPeer implements DiscoveryListener {

    private PeerGroup netpg = null; // NetPeergroup
    private PeerGroup restoNet = null; // RestoNet peergroup

    // Services within the RestoNet peergroup
    private DiscoveryService disco;  // Discovery service
    private PipeService pipes;       // Pipe service
    private PipeAdvertisement myAdv; // HungryPeer pipe advertisement
    private InputPipe myPipe;        // Input pipe to HungryPeer

    private MimeMediaType mimeType = new MimeMediaType("text", "xml");

    private int time-out = 3000;         // Discovery time-out
    private int rtime-out = 30000;       // Pipe resolver time-out
    // All RestoPeers found
    private Vector restoPeerAdvs = new Vector();
    private Vector restoPeerPipes = new Vector();

    private String myIdentity = "Bill Joy"; // My identity
    private String friesRequest = "medium"; // Fries auction request

    public static void main(String args[]) {
        HungryPeer myapp = new HungryPeer();
        myapp.startJxta();
        System.exit(0);
    }

    private void startJxta() {
        try {
            // Discover (or create) and join the default JXTA NetPeerGroup
            netpg = PeerGroupFactory.newNetPeerGroup();
        } catch (PeerGroupException e) {
            // Couldn’t initialize; can’t continue
            System.out.println("Fatal error : creating the NetPeerGroup");
            System.exit(1);
        }

        // Discover and join the RestoNet peergroup
        try {
            if (!joinRestoNet()) {
                System.out.println("Error joining the RestoNet peergroup");
            }
        } catch (PeerGroupException e) {
            System.out.println("Error joining the RestoNet peergroup");
            System.exit(1);
        }
    }
}

private boolean joinRestoNet() {
    // Should return true if successful
    return true;
}
"Sorry could not find the RestoNet Peergroup";
    System.exit(2);
}
} catch (Exception e) {
    System.out.println("Can’t join RestoNet group");
    System.exit(1);
}

// Set our HungryPeer communication pipe so RestoPeers can talk to us
if (!setHungryPeerPipe()) {
    System.out.println("Abort due to failure to create our HungryPeer pipe");
    System.exit(1);
}

// Attempt to locate RestoPeers in RestoNet
discoverRestoPeers();

// Connect to RestoPeers that have been discovered
connectToRestoPeers();

// I am hungry. Send an auction request for French fries
// to the connected RestoPeers.
sendFriesAuctionRequests();

// Process incoming bids from RestoPeers
receiveFriesBids();

// This method is used to discover the RestoNet peergroup
// If found, the peer will join the peergroup
private boolean joinRestoNet() {
    // Same as example in Chapter 4
}

// Create the HungryPeer pipe to receive bid responses from RestoPeers.
// The advertisement of this pipe is sent as part of the auction request
// for RestoPeers to respond.
private boolean setHungryPeerPipe() {
    try {
        // Create a pipe advertisement for our HungryPeer.
        // This pipe will be used within the RestoNet peergroup
        // for other peers to talk to our HungryPeer.
        myAdv = (PipeAdvertisement) AdvertisementFactory.newAdvertisement(
                PipeAdvertisement.getAdvertisementType());
    } catch (Exception e) {
        System.out.println("Can’t create HungryPeer pipe");
        System.exit(1);
    }
    return true;
}
// Initialize the advertisement with unique peer information
// so we can communicate
myAdv.setPipeID(IDFactory.newPipeID(
    restoNet.getPeerGroupID(  )))
myAdv.setName("restoNet:HungryPipe:" + myIdentity);

// Set the pipe type to be unicast unidirectional
myAdv.setType(PipeService.UnicastType);

// Create the input pipe
myPipe = pipes.createInputPipe(myAdv);
} catch (Exception e) {
    System.out.println("Could not create the HungryPeer pipe");
    return false;
} return true;

// Discover RestoPeers that have joined RestoNet
// RestoPeers are discovered via their published pipe advertisement
private void discoverRestoPeers(  ) {
    int found = 0;           // Number of RestoPeers found
    int count = 10;          // Discovery retries
    System.out.println("Locating RestoPeers in the RestoNet Peergroup");

    // Try to find at least two RestoPeers (an arbitrary number)
    // RestoPeers are found by their pipe advertisements
    while (count-- >0) {
        try {
            // Check if we already have restaurant advertisements
            // in our local peer cache
            Enumeration ae = disco.getLocalAdvertisements(DiscoveryService.ADV, "name", "RestoNet:RestoPipe:*");

            // If we found some advertisements in the cache,
            // add them to the list
            if (ae != null && ae.hasMoreElements(  )) {
                // Reset count and restoPeerAdvs as we have just previously
                // retrieved all advertisements, including cached ones
                found = 0;
                restoPeerAdvs.removeAllElements(  );
            }
        } catch (Exception e) {
            System.out.println("Could not discover the RestoPeers");
            return false;
        }
    }
}
while (ae.hasMoreElements()) {
    restoPeerAdvs.addElement(ae.nextElement());
    ++found;
    if (found > 1) break;  // Want to find at least two
}

// Did not find enough advertisements in the cache. Send a remote discovery request to search for more RestoPeer advertisements.
disco.getRemoteAdvertisements(null, DiscoveryService.ADV, "name", "RestoNet:RestoPipe:**", 5, null);

// Give the peers a chance to respond
try {
    Thread.sleep(time-out);
} catch (InterruptedException e) {}

} catch (IOException e) {
    // Found nothing! Move on.
}

} // Completed RestoPeer discovery
System.out.println("Found \n + found + 
RestoPeers(s)");

} // Method to connect and open output pipes to all the RestoPeers that we have discovered. Each RestoPeer is identified by its unique RestoPeer pipe advertisement.
private void connectToRestoPeers() {
    // Enumerate all the RestoPeer pipe advertisements we have discovered
    // and attempt to connect a pipe to each of them
    for (Enumeration en = restoPeerAdvs.elements(); en.hasMoreElements();)
        PipeAdvertisement padv = (PipeAdvertisement) en.nextElement();
    try {
        System.out.println("Attempting to connect to discovered RestoPeer");
        OutputPipe pipeOut = pipes.createOutputPipe(padv,
// Check if we have a connected pipe
if (pipeOut == null) {
    // Failed; go to next RestoPeer
    System.out.println("Failure to connect to RestoPeer Pipe:" +
                        padv.getName());
    continue;
}
// Save the output pipe
restoPeerPipes.addElement(pipeOut);
System.out.println("Connected pipe to " + padv.getName());
} catch (Exception e) {
    // Error during connection; go to next RestoPeer
    System.out.println("RestoPeer may not be there anymore:" +
                        padv.getName());
    continue;
}

private void sendFriesAuctionRequest() {
    // See later example
}

private void receiveFriesBids() {
    // See later example
}

We've enhanced the startJxta() method so that in addition to joining the RestoNet peergroup, it now:

- Creates a pipe over which it can receive bids
- Discovers and then connects to RestoPeers
- Sends a request to the RestoPeers
- Processes the responses it gets back

For now, we're focusing on how it creates its pipe and how it discovers and connects to the RestoPeers.

### 5.3.1 Creating the Pipe Advertisement

Let's start with the setHungryPeerPipe() method, which creates the pipe over which the HungryPeer receives bids. To create the pipe, follow the same procedure used by the RestoPeer: create a pipe advertisement and set the ID, type, and name of the pipe using the Pipe service of the RestoNet peergroup to create the pipe. Later, this pipe advertisement may be sent as part of the auction request to the RestoPeer. Note that this means the advertisement does not need to be published.
The only difference in creating the pipe is the name we use to construct it. Ideally, this name should be unique, but this cannot be guaranteed unless a centralized naming service is used. The HungryPeer class uses "RestoNet:HungryPeer:Bill Joy" as its name; you should change this name before you run this example.

### 5.3.2 Discovering the Peer Pipe Advertisement

The `discoverRestoPeers()` method is used to find the RestoPeers that are currently members of the RestoNet peergroup; the HungryPeer is interested only in restaurants that are running and can process auction requests. The `discoverRestoPeers()` method uses the discovery service of the RestoNet peergroup to locate RestoPeers. HungryPeers discover RestoPeers via the pipe advertisement that each RestoPeer publishes. Discovering the pipe advertisement follows the same general principle we've used to discover any resource: we check the local cache, and if we're not satisfied, we send a remote request and wait for responses to come in.

There are a few differences in the actual implementation, however. The name of the pipe advertisement that we're looking for is "RestoNet:RestoPipe*". (The wildcard "*" character is used to match any string.) This allows us to find multiple RestoPeers based on the common formats of the names of their pipe advertisements.

A second difference is in how we process matching advertisements. If advertisements are found in the local cache, we check to see how many we found. The `discoverRestoPeer()` method is written to attempt to discover at least two RestoPeers. This is just an arbitrary example: we might have picked a different number, or we might have chosen to terminate our search based on some other criteria. There is no guarantee that 2 RestoPeers will be found, even after 10 attempts at discovery. When we do find at least 2 RestoPeer pipe advertisements in the cache, or when we've tried 10 times, we might decide that enough is enough and exit from the discovery loop.

Note that if we exit after 10 attempts, we may have found 0, 1, or 2 RestoPeers.

The desire to find multiple peers triggers another change in our basic code: in the `getRemoteAdvertisements()` method, the threshold argument is now set to 5. We needed to increase this number, since we're interested in receiving multiple responses (though this argument is not used in JXTA 1.0, it may in the future limit how many responses are placed into the local cache).

### 5.3.3 Connecting to the Remote Peer

The `connectToRestoPeers()` method is used to connect a pipe to the discovered RestoPeers. The `connectToRestoPeers()` method enumerates all discovered pipe advertisements stored in `restoPeerAdvs` vector and attempts to create an output pipe endpoint for each of them. The `createOutputPipe()` method of the RestoNet pipe service creates and resolves the output pipe endpoint. As before, in order to create the output pipe, we must supply the pipe advertisement (in this case, the pipe advertisement of the RestoPeer) and the time-out for creating the pipe (30 seconds).

Note that creating the output pipe may fail: having found a RestoPeer pipe advertisement does not guarantee that the corresponding RestoPeer is up and listening on the pipe. Therefore, we simply print a warning message when the pipe creation fails, and we continue on to the next RestoPeer. When we're done, the `restoPeerPipes` vector contains all of the pipes to which we've successfully connected.

### 5.3.4 Sending the Auction Request

Once we've discovered and connected to the RestoPeers, we're ready to construct our auction request and send it to each of them. This is done with the `sendFriesAuctionRequests()` method, as follows:

```java
// Send an auction request for French fries to all the RestoPeer
```
The `sendFriesAuctionRequests()` method sends an auction request message to all of the connected RestoPeer pipes. It follows the basic procedure to create a message: the structured document is created from the `StructuredDocumentFactory` class, and the appropriate
"Name" and "Fries" elements are added to the request. Then the request and the previously created pipe advertisement are added to the message envelope, which is then sent to all RestoPeers that were stored in the `restoPeerPipes` vector.

5.3.5 Receiving Big Messages

Finally, our HungryPeer can process any responses it receives to its auction request; it does so with the `receiveFriesBids( )` method:

```java
private void receiveFriesBids( ) {
    // Continue until we get all answers
    while (true) {
        Message msg = null; // Pipe message received
        String price = null; // Fries price bid
        String brand = null; // RestoPeer name offering the bid
        String specials = null; // Specials offer bid
        InputStream ip = null; // Input stream to read msg element
        StructuredDocument bid = null; // Bid document received

        try {
            // Wait for a bid message to arrive from a RestoPeer
            msg = myPipe.waitForMessage( );

            // Check if the message is valid
            if (msg == null) {
                if (Thread.interrupted( )) {
                    // We have been asked to stop
                    System.out.println("Abort Receiving bid loop interrupted");
                    myPipe.close( ); // Close the pipe
                    return;
                }
            }

            if (msg == null) {
                catch (Exception ex) {
                    // Error in receiving message
                    myPipe.close( );
                    System.out.println("Abort Receiving Error receiving bids");
                    return;
                }

                // We got a message from a RestoPeer. Extract and display information about the bid received.
                try {
                    // Extract the bid document from the message
```
ip = msg.getElement("Bid").getStream();

bid = StructuredDocumentFactory.newStructuredDocument(
    mimeType, ip);

// Parse the document to extract bid information
Enumeration enum = bid.getChildren();
while (enum.hasMoreElements()) {
    Element element = (Element) enum.nextElement();
    String attr = (String) element.getKey();
    String value = (String) element.getValue();
    if (attr.equals("Price")) {
        price = value;
        continue;
    }
    if (attr.equals("Brand")) {
        brand = value;
        continue;
    }
    if (attr.equals("Specials")) {
        specials = value;
        continue;
    }
}

// We got a valid bid. Print it.
System.out.println("Received Fries Bid from RestoPeers (" + brand + ") at a Price ($" + price + ") \nRestoPeers Special (" + specials + ")");

} catch (Exception e) {
    // Broken content
    System.out.println("Error extracting bid from the message");
    continue;
}

}

The receiveFriesBids() method waits for messages on the HungryPeer input pipe endpoint created earlier by the setHungryPeerPipe() method. Since a single input pipe endpoint can bind to many output pipe endpoints, all bid offers can be received from the same pipe: we have an n-to-1 pipe connection.

When a valid message arrives, the bid offer is processed via techniques that are familiar: sections of the message are retrieved via the getElement() method and converted into structured documents. Then tags are retrieved from the structured document, and we print out each bid that we receive.

There is no guarantee that a bid offer will be received for every auction request that was sent. So after waiting for some period of time, the HungryPeer can select the cheapest RestoPeer bid (perhaps taking advantage of a special) and place its order. Since this logic doesn't involve the JXTA platform, we leave it as an exercise for the reader.

5.4 Running the Pipe Example
Once again, we must set up separate directories for the RestoPeer and the HungryPeer. After compiling the code, we can run it.

Let's run the RestoPeer first and see what happens. After the platform initializes, we get this output:

```
Attempting to Discover the RestoNet PeerGroup
Could not find the RestoNet peergroup; creating one
RestoNet Restaurant (Chez JXTA) is on-line
Attempting to Discover the Restaurant RestoPipe
Could not find the Restaurant Pipe Advertisement
Created the Restaurant Pipe Advertisement
RestoNet Restaurant (Chez JXTA) waiting for HungryPeer requests
```

The RestoPeer instance (Chez JXTA) tries to find the RestoNet peergroup. Assuming you are on an isolated network and are running this from a new directory, it cannot find the existing peergroup, so it creates one. The RestoPeer then tries to discover its pipe advertisement. Again, since this is running in a new directory, the RestoPeer pipe advertisement is not found; the RestoPeer creates and publishes it and then waits for HungryPeer requests.

Let's start running the HungryPeer instance:

```
Attempting to discover the RestoNet Peergroup
Found the RestoNet PeerGroup Advertisement
The HungryPeer joined the restoNet PeerGroup
Locating RestoPeers in the RestoNet Peergroup
Found 1 RestoPeers(s)
Attempting to connect to discovered RestoPeer
Connected pipe to RestoNet:RestoPipe:Chez JXTA
Sent Fries Auction Request (medium) to connected peers
Received Fries Bid from RestoPeers (Chez JXTA) at a Price ($2.50)
RestoPeers Special (large ($3.00))
```

As before, the HungryPeer joins the RestoNet peergroup; now it tries to locate RestoPeers that have also joined RestoNet. One RestoPeer is found (Chez JXTA). Since there's only 1 RestoPeer on our isolated network, this process takes a while: we must wait for the discovery loop to complete all 10 iterations before we give up (having found only 1 peer). The HungryPeer indicates that it sends a request for medium fries to the RestoPeers; it receives one answer indicating a price of $2.50 and special offer of large fries for $3.00.

If we look back at the output of the RestoPeer, new messages have been printed:

```
Received Request from HungryPeer Bill Joy for medium Fries.
Attempting to create Output Pipe to HungryPeer Bill Joy
Sent Bid Offer to HungryPeer (Bill Joy) Fries price = 2.50, special = large ($3.00)
```

These messages indicate that the RestoPeer received an auction request from our HungryPeer "Bill Joy" for medium French fries. The RestoPeer resolved the pipe connection to the HungryPeer and sent a bid offer at $2.50 with the special RestoPeer offering.

Of course, you may encounter different behaviors when running this example if there are other JXTA peers on your network: there may be multiple instances of RestoPeers and HungryPeers running on the network. You may be lucky and receive more bid offers! In any case, you may want to modify the names of the pipes in the RestoPeer and HungryPeer and run multiple instances of each of them to see what happens. For example, if we run two RestoPeers (Chez JXTA and Bistro Frites), the HungryPeer log will look like the following:

```
Attempting to discover the RestoNet Peergroup
Found the RestoNet PeerGroup Advertisement
The HungryPeer joined the restoNet PeerGroup
Locating RestoPeers in the RestoNet Peergroup
Found 2 RestoPeers(s)
Attempt to connect to discovered RestoPeer
Connected pipe to RestoNet:RestoPipe:Chez JXTA
```
Attempt to connect to discovered RestoPeer
Connected pipe to RestoNet:RestoPipe:Bistro Frites
Sent Fries Auction Request (medium) to connected peers
Sent Fries Auction Request (medium) to connected peers
Received Fries Bid from RestoPeers (Chez JXTA) at a Price ($2.50)
RestoPeers Special (large ($3.00))
Received Fries Bid from RestoPeers (Bistro Frites) at a Price ($2.25)
RestoPeers Special (Medium ($2.25))

The HungryPeer found two RestoPeers, connected to them, and received two bid offers: one at $2.50 and the other at $2.25. 5.5 Event-Based Programming

In our examples so far, we’ve used synchronous methods to obtain all of our information. When we discovered peergroups or pipe advertisements, we sent the discovery message and then synchronously examined the local cache to see if the resource had been located. When we read messages from the pipe, we blocked until a message was available. Behind the scenes, of course, discovery was asynchronous, but the threads listening for events in that process were hidden to us: all that we invoked were synchronous methods.

However, there is another mechanism available for these calls: an asynchronous, event-based API. This mechanism follows Java’s standard event design pattern: you register a listener class to be informed when certain events occur. When these events occur, methods of the listener class are invoked.

We’ll show how this works by modifying our HungryPeer to use event-based discovery to locate the RestoNet pipes. Here’s the code that does this:

```java
import java.io.*;
import java.util.Enumeration;
import java.util.Vector;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.exception.PeerGroupException;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.Advertisement;
import net.jxta.document.StructuredDocument;
import net.jxta.document.Element;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.discovery.DiscoveryService;
import net.jxta.pipe.PipeService;
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.PipeID;
import net.jxta.pipe.OutputPipe;
import net.jxta.endpoint.Message;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;
import net.jxta.id.IDFactory;
import net.jxta.discovery.DiscoveryEvent;
import net.jxta.discovery.DiscoveryListener;
import net.jxta.protocol.DiscoveryResponseMsg;
```
public class HungryPeer implements DiscoveryListener {

    private PeerGroup netpg = null;  // NetPeergroup
    private PeerGroup restoNet = null;  // RestoNet peer group

    // Services within the RestoNet peer group
    private DiscoveryService disco;  // Discovery service
    private PipeService pipes;  // Pipe service
    private PipeAdvertisement myAdv;  // HungryPeer pipe advertisement
    private InputPipe myPipe;  // Input pipe to talk to HungryPeer
    private MimeMediaType mimeType = new MimeMediaType("text", "xml");

    private int time-out = 3000;  // Discovery time-out
    private int rtime-out = 30000;  // Pipe resolver time-out

    // All RestoPeers found
    private Vector restoPeerAdvs = new Vector();
    private Vector restoPeerPipes = new Vector();

    private String myIdentity = "Bill Joy";  // Identity of this peer
    private String friesRequest = "medium";  // Fries auction request

    public static void main(String args[]) {
        HungryPeer myapp = new HungryPeer();
        myapp.startJxta();
        System.exit(0);
    }

    private void startJxta() {
        try {
            // Discover (or create) and join the default JXTA NetPeerGroup
            netpg = PeerGroupFactory.newNetPeerGroup();
        } catch (PeerGroupException e) {
            // Couldn’t initialize; can’t continue
            System.out.println("Fatal error : creating the NetPeerGroup");
            System.exit(1);
        }

        // Discover and join the RestoNet peer group
try {
  if (!joinRestoNet()) {
    System.out.println("Sorry could not find the RestoNet PeerGroup");
    System.exit(2);
  }
}

// Set our HungryPeer communication pipe so RestoPeers can talk to us
if (!setHungryPeerPipe()) {
  System.out.println("Aborting due to failure to create our HungryPeer pipe");
  System.exit(1);
}

// Register for discovery events for pipe advertisements
disco.addDiscoveryListener(this);
disco.getRemoteAdvertisements(null, DiscoveryService.ADV,
  "name", "RestoNet:RestoPipe:*", 5, null);

// Wait; processing events as they happen
while(true) {
  receiveFriesBids();
}

public void discoveryEvent(DiscoveryEvent ev) {
  System.out.println("Processing discovery event");
  DiscoveryResponseMsg msg = ev.getResponse();
  // The enumeration contains all the pipe advertisements that were found
  Enumeration e = msg.getResponses();
  while (e.hasMoreElements()) {
    try {
      String s = (String) e.nextElement();
      PipeAdvertisement adv = (PipeAdvertisement)
        AdvertisementFactory.newAdvertisement(
          new MimeMediaType("text/xml"),
          new ByteArrayInputStream(s.getBytes()));
      connectAndSend(adv);
    } catch (Exception ex) {
      System.out.println("Can’t connect to peer "+ ex);
      continue;
    }
  }
}
// This method is used to discover the RestoNet peergroup
// If found, the peer will join the peergroup
private boolean joinRestoNet( ) {
    int count = 3; // Maximum number of attempts to discover
    System.out.println("Attempting to discover the RestoNet Peergroup");
    // Get the discovery service handle from the NetPeerGroup
    DiscoveryService hdisco = netpg.getDiscoveryService();
    // All discovered RestoNet peers
    Enumeration ae = null;
    // Loop until we find the "RestoNet" peergroup advertisement
    // or we’ve exhausted the desired number of attempts
    while (count-- > 0) {
        try {
            // Check if we have the advertisement in the local
            // peer cache
            ae = hdisco.getLocalAdvertisements(
                DiscoveryService.GROUP, "Name", "RestoNet");
            // If we found the RestoNet advertisement, we are done
            if ((ae != null) && ae.hasMoreElements( )) break;
            // The RestoNet advertisement is not in the local
            // cache. Send a discovery request to search for it.
            hdisco.getRemoteAdvertisements(null, DiscoveryService.GROUP, "Name", "RestoNet", 1, null);
            // Wait to give peers a chance to respond
            try {
                Thread.sleep(time-out);
            } catch (InterruptedException ie) {} catch (IOException e) {
                // Found nothing! Move on.
            }
        } catch (IOException e) {
            // Found nothing! Move on.
        }
    }
}
// Check if we found the RestoNet advertisement
if (ae == null || !ae.hasMoreElements()) {
    return false;
}

System.out.println("Found the RestoNet PeerGroup Advertisement");
// Get the advertisement
PeerGroupAdvertisement adv = (PeerGroupAdvertisement) ae.nextElement();

try {
    // Call the peergroup factory to instantiate a new peergroup instance
    restoNet = netpg.newGroup(adv);
    // Get the discovery and pipe services to use within the RestoNet peergroup
    disco = restoNet.getDiscoveryService();
pipes = restoNet.getPipeService();
} catch (Exception e) {
    System.out.println("Could not create RestoPeerGroup");
    return false;
}

System.out.println("The HungryPeer joined the restoNet PeerGroup");
return true;

// Create the HungryPeer pipe to receive bid responses from RestoPeers.
// The advertisement of this pipe is sent as part of the auction request
// for RestoPeers to respond to.
private boolean setHungryPeerPipe() {
    try {
        // Create a pipe advertisement for our HungryPeer.
        This pipe will be used within the RestoNet peergroup for other peers to talk to our hungry peer.
        myAdv = (PipeAdvertisement) AdvertisementFactory.newAdvertisement(PipeAdvertisement.getAdvertisementType());
        myAdv.setPipeID(IDFactory.newPipeID(restoNet.getPeerGroupID()));
        myAdv.setName("restoNet:HungryPipe:" + myIdentity);
        // Initialize the advertisement with unique peer information
    // so we can communicate
    myAdv.setPipeID(IDFactory.newPipeID(restoNet.getPeerGroupID()));
    myAdv.setName("restoNet:HungryPipe:" + myIdentity);
// Set the pipe type to be unicast unidirectional
myAdv.setType(PipeService.UnicastType);

// Create the input pipe
myPipe = pipes.createInputPipe(myAdv);
} catch (Exception e) {
    System.out.println("Could not create the HungryPeer pipe");
    return false;
}
return true;

// Method to connect and open output pipes to all of the RestoPeers that we have discovered. Each RestoPeer is identified by its unique RestoPeer pipe advertisement. This method is essentially the same as the connectToRestoPeers() and sendFriesRequests() methods in our previous example.
private void connectAndSend(PipeAdvertisement padv) throws Exception {
    System.out.println("Attempting to connect to discovered RestoPeer");

    // Create an output pipe connection to the RestoPeer
    OutputPipe op = pipes.createOutputPipe(padv, rtimeout);

    // Check if we have a connected pipe
    if (op == null) {
        // Failed; go to next RestoPeer
        System.out.println("Failure to connect to RestoPeer Pipe:" + padv.getName());
        return;
    }

    StructuredDocument request = StructuredDocumentFactory.newStructuredDocument(mimeType, "RestoNet:Request");

    // Fill in the Fries auction request argument
    Element re;
    re = request.createElement("Name", myIdentity);
    request.appendChild(re);
    re = request.createElement("Fries", friesRequest);
    request.appendChild(re);

    // Create the pipe message to send
    Message msg = pipes.createMessage();
// Fill in the first message element, which is the HungryPeer pipe advertisement return address. We need this so RestoPeers can respond to us.
msg.addElement(msg.newMessageElement("HungryPeerPipe", mimeType, myAdv.getDocument(mimeType).getStream( )));

// Fill in the second message element, which is the fries request.
// Insert the document in the message.
msg.addElement(msg.newMessageElement("Request", mimeType, request.getStream( )));

// Send the auction message to the RestoPeer
op.send(msg);
System.out.println("Sent Fries Auction Request (" + friesRequest + ") to connected peers");
}

private void receiveFriesBids( ) {
    // Identical to previous example
}

For event-based programming, the first thing we must do is create a listener that can be registered to receive the appropriate event. The listener interface that handles discovery events is the DiscoveryListener interface (net.jxta.discovery.DiscoveryListener); we've implemented this interface in the new HungryPeer class. This interface has a single method that we must implement: the discoveryEvent( ) method. The listener is registered with the discovery service by calling the addDiscoveryListener( ) method. After this method is called, the discoveryEvent( ) method will be called whenever the discovery service locates a new JXTA resource. In order to distinguish between resources, the remotePublish( ) method returns an ID that is part of the discovery event. Therefore, when we publish the request to discover RestoNet pipes, we save the ID so that when we get an event, we can check its ID and see if it's the event we want. Note that we must still call the remotePublish( ) method in order for the discovery service to start the discovery protocol for that resource. When one or more RestoNet pipe advertisements are discovered, they are packed into a DiscoveryEvent object (net.jxta.discovery.DiscoveryEvent) and asynchronously passed to the discoveryEvent( ) method. The enumeration of these pipe advertisements is returned from the getResponses( ) method; we cycle through the enumeration and create the pipe advertisement from the XML document that the event object carries. Once we have the pipe advertisement, we treat it just like we would RestoNet pipe advertisements: we create the pipe for the advertisement and send the request over the pipe. The code to do this is in the connectAndSend( ) method, which is essentially a conflation of our previous connectToRestoPeers( ) and sendFriesRequests( ) methods (except that now we're sending only to a single pipe). When the response from the RestoPeer comes back, it is processed by the receiveFriesBids( ) method, which is unchanged from our previous example.

This type of event-driven programming is available for the pipes themselves. Instead of issuing a blocking waitForMessage( ) method on the pipe, we can create an object that implements the PipeMsgListener interface (jxta.net.pipe.PipeMsgListener) and register that object with the pipe service. The pipeMsgEvent( ) method of our listener would be called when a message is received; the message would be retrieved from the PipeMsgEvent
via the `getMessage()` method, and the message would be processed exactly like we processed messages received from the `waitForMessage()` method. We leave the full implementation of this code to the reader.
You may wonder at this point whether the synchronous approach we started with is superior or inferior to this asynchronous approach. These is no one approach that is always valid, which is why the JXTA APIs provide both solutions. In the case of our HungryPeer, the synchronous approach seems like a better fit: if we're hungry, we're really interested only in who is responding now, so we're better off waiting for 30 seconds to see who will sell us some French fries, and dealing only with those peers.
On the other hand, if we're buying a new flat-panel display for our desktop computer, we wouldn't care so much about the immediacy of the responses; we might want to wait around for a week or so to see who has a special offer. In this case, we're better off registering for events: when new FlatPanelSellerPeers come online, we'll be notified about them and can look at their offers and make a decision at that time.
In general, program design issues will determine which approach is better for a particular application. JXTA provides the flexibility for you to use either approach.

### 5.6 Summary

In this chapter, we saw how JXTA peers use pipes to communicate between themselves. Pipes are a logical abstraction that hide the complexity of network protocols and topology from peers. Peers need to publish only their pipe advertisements and obtain pipe advertisements from other peers; the core pipe service is used to resolve the underlying network connections represented by these advertisements. Therefore, peers can focus on reading and interpreting the messages they send over the pipes: the “what” of their communication rather than the “how.” We’ve also seen how event-based, asynchronous APIs can be used to process newly discovered advertisements and newly available pipe messages.

JXTA pipes enable two peers to communicate independently of their physical location on the network. JXTA pipes can traverse firewalls and NAT systems. Two peers, each behind its own firewall, can exchange messages through a JXTA pipe. JXTA’s pipe advertisements virtualize the notion of the peers’ location.

In this chapter, our two peers were applications, and a HungryPeer depended on an available RestoPeer in order to satisfy its requests. In the next chapter, we’ll see how a JXTA application can be a JXTA service. A JXTA service can be created by any application that needs to use it; applications do not need to rely on finding an instance of the service within the peergroup.
Chapter 6. JXTA Network Services

In this chapter, we discuss how network services are created, published, and discovered by the JXTA platform. We use the term "network service" to represent any kind of service (web services, legacy services, CORBA services, RMI services, etc.) available on the network. JXTA is as agnostic as possible regarding the service invocation model used to access a network service. JXTA's pipe construct can create services using pipes as the principal invocation mechanism. But pipes are not required for all network services. Upcoming standards such as WSDL, ebXML, SOAP, and UPnP may be used by a JXTA application to invoke a service once the location of that service has been discovered.

A JXTA application can use a SOAP connection to a WSDL service, an RMI connection to contact a remote server object, or a pipe connection to communicate with a JXTA service. The JXTA platform does not impose any restrictions on the service invocation model used. However, the underlying peer infrastructure will ultimately dictate which service invocation model is used. For example, if a Java runtime environment is not available, an application will not be able to activate a RMI-based service via an RMI call.

The JXTA protocols do not contain a specific service invocation protocol by design. This was done to enable a JXTA application to interoperate with a wide range of services. Due to the large variety of service invocation models and the lack of standards in that area, it was thought to be more practical to leave service invocation outside the scope of JXTA. However, services that use pipes will have better integration with the JXTA platform.

There are two fundamental kinds of network services in the JXTA platform: peer services and peergroup services. We start this chapter with an explanation of each type of service; following this, we'll show the changes required in order for our RestoPeer and HungryPeer to conform to the JXTA network service model. We'll conclude with a discussion of how an arbitrary, non-JXTA application can be turned into a JXTA service.

6.1 JXTA Services

The JXTA platform provides an infrastructure to publish and discover network services via the PDP. Like any JXTA resource, a network service is represented by a service advertisement: an XML document that contains all of the information that uniquely identifies a service and all of the information necessary to invoke a service. Members of a peergroup can publish and discover service advertisements just as they operate on any other advertisements.

A service is made available to members of a peergroup by publishing a service advertisement in the peergroup. The JXTA platform provides a distinction between two types of services; depending on how the service is published in the peergroup, a JXTA service is either a peer service or a peergroup service.

The difference is shown in Figure 6-1. All services are available within the peergroup, but they run differently. Peer services are instantiated by the peer that provides the service. Peergroup services are instantiated by each peer. We've seen peergroup services in many of our examples: the core services (the discovery service, the pipe service, and so on) are all peergroup services.

Figure 6-1. Peer and peergroup services
6.1.1 Peer Services

A peer service is instantiated only on the peer that publishes the service. Multiple instances of the service may run within a peergroup if multiple peers decide that they want to provide the service; each peer will advertise the service separately. A peer service is advertised by publishing a service advertisement throughout the peergroup. Members of the peergroup discover the peer service by searching for the service advertisement. The service advertisement contains all of the information necessary to invoke the service. The service is unavailable until at least one peer instantiates and advertises the service; if this peer becomes unavailable, the service will fail. This is one reason why it’s good to have multiple peers that run the service: service availability is increased. A peer can advertise a peer service any time after it joins the peergroup and can publish the same service in multiple peergroups (as long as it belongs to each peergroup, of course). Members of the peergroup can discover peer services at any time.

6.1.2 Peergroup Services

A peergroup service is composed of a collection of service instances (potentially cooperating with each other) running on multiple members of a peergroup. If any one peer fails, the collective peergroup service is not affected because chances are, the service is still available from another peer member. Peergroup services are advertised as part of the peergroup advertisement. The main differences between a peer service and a peergroup service are in how the service is advertised and how it is instantiated. A peer service is instantiated only on the peer that advertises the service. A peergroup service is potentially instantiated on all of the peer members of the peergroup. The decision to instantiate a peergroup service is left to the peer and the peergroup implementation. Some peergroups may require that every member instantiate all peergroup services; other peergroup implementations will leave that decision to the peer. In general, though, each peer will implement the peergroup services that it needs to use. Peergroup services are used by every member of the peergroup. For example, a content-sharing peergroup may have a content-sharing peergroup service, while a chat peergroup may have a chatroom service; these are peergroup services because each peer in the peergroup will use these services. The most common peergroup services are the core peergroup services that we’ve been discussing all along: the membership, discovery, resolver, pipe, peer information, and rendezvous services.

6.1.3 Why Services?

We’ve seen how a JXTA peer can set up a pipe advertisement and makes its functionality available to members of a peergroup. Why would it choose to advertise itself as a peer or peergroup service instead? Because there are several advantages to being a service, including:
Service instantiation
Services can be either preinstalled into a peer or loaded from the network. The process of finding, downloading, and installing a service from the network is similar to performing a search on the Internet for a web page, retrieving the page, and then installing the required plug-in.
In order to run a service, a peer may have to locate an implementation suitable for the peer runtime environment. Multiple implementations of the same service may allow Java peers to use Java code implementations, C++ peers to use C++ implementations, and so on. The service advertisement contains a URL to locate suitable implementations of a service as well as all other necessary information to instantiate the service.

Moving behaviors
Service advertisements provide a simple framework to describe a network resource; they enable the resource to be discovered and moved. At any point in time, a peer may discover a peer service advertisement and decide to instantiate a new instance of the service on this peer. The instantiation of peer services can be done at any time: when the advertisement is first discovered, when the service is first needed, and so on. The JXTA 1.0 implementation instantiates all peergroup core services at the time the peer joins the peergroup (one possible implementation). The location of the service will change over time, but since the service is located dynamically, this is not a problem.

Service reliability and availability
The ability to replicate services within a peergroup is essential to building widely available services. Replicating a service within a peergroup increases the availability of that service within a peergroup. Since replication is usually random, it is very difficult to perform denial of service attacks on a JXTA service. There are no centralized structures keeping track of which peers are running an instance of the service. As more peers join the peergroup, more peers are available to instantiate the service, which increases the overall availability of the service.
JXTA-enabled services provide the added capability of the dynamic-binding pipe mechanism. The pipe-binding feature enables an application to easily bind a pipe connection to another instance of a service.
Services that require tightly coupled synchronization between each instance have more difficulty being deployed in a P2P environment. Such services require tightly coupled peergroups that act very much like highly available clusters. It is more expensive to replicate an instance of such a service, since synchronization messages need to be exchanged between instances. Peergroups with such services are much less scalable than other peergroups.

Dynamic behaviors
JXTA services are a concrete representation of JXTA modules, which allow service definitions and behaviors to be defined dynamically within JXTA peers.

6.2 JXTA Modules
JXTA has made the deliberate decision to specify mechanisms, rather than policies, for building P2P applications. For example, the JXTA platform implementation does not impose a unique search or authentication policy on JXTA developers. Each developer can specify the most appropriate authentication or search policies within the scope of his peergroups. Policies may be shared between peergroups when the same behavior is required. The JXTA peergroup framework provides the necessary mechanisms via the peergroup advertisement for developers to configure the set of policy behaviors used within each peergroup. The JXTA platform's open-policy approach allows a multitude of behaviors for supporting a large variety of application domains. For instance, authentication policies for a financial-transaction peergroup are likely to be different than in a content-sharing peergroup, in which content accesses are free.
JXTA is designed to be independent of programming languages, system platforms, or service invocation models. JXTA embraces a heterogeneous world in which devices running different platforms and service invocation models can discover and interact with each other. Therefore, JXTA policy behaviors need to be described in an independent manner.
JXTA also believes in a dynamic world where peers can instantiate new policy behaviors as they interact with other peers and discover new peergroup behaviors. This viral effect enables peers to dynamically load and unload new behaviors to meet the requirements of the current peergroup’s operating environment. For example, a peer joining a peergroup using a specific authentication mechanism (such as Kerberos) can dynamically load an (Kerberos) authenticator that implements the required authentication policy when the authenticator is not yet available on that peer. However, policies are not required to be downloaded; some may be preloaded on a peer. For instance, all of the NetPeerGroup policy behaviors are preloaded on the JXTA platform. Tiny devices, such as single-function devices (e.g., beepers), also may not have the capability to load new behaviors, and are likely to have all of their policy behaviors statically configured.

At the center of the independent and policy-agnostic JXTA platform is the concept of modules. Modules are used for representing pieces of code that can be dynamically loaded and instantiated on a peer to implement a new behavior. Applications, network services, and peergroup policies are examples of behaviors that can be instantiated on a peer, represented by modules. The JXTA module abstraction does not impose or specify what the "code" representation is. The code representation can be a Java class, a Java jar file, a dynamic library (DLL), a choreography set of XML messages (WSDL), or a script. Multiple implementations of a module may exist to support various runtime environments. For example, a module can be implemented on different platforms, such as the Java platform, Microsoft Windows, or the Solaris Operating Environment. Modules provide a generic abstraction to allow peers to load and instantiate new behaviors. As peers browse or join peergroups, they may find new behaviors that they are interested in or are required to instantiate. For example, when joining a peergroup, a peer may have to instantiate a new search service.

The JXTA module framework enables the representation and advertisement of platform-independent modules and allows peers to instantiate any type of implementation. For example, a peer has the ability to instantiate either a Java or a C implementation of a module's function of its runtime characteristics. The ability to describe and publish platform-independent modules is essential to support peergroups composed of heterogeneous peers. The JXTA platform uses modules for self-description and for describing the default NetPeerGroup.

As with any resource in the JXTA network, modules are represented by advertisements. Three module advertisements are used: a module class, a module specification, and a module implementation.

### 6.2.1 Module Class Advertisement

The module class advertisement is used for advertising the existence of a module in the JXTA network; it is represented in the Java bindings by the `ModuleClassAdvertisement` class (`net.jxta.protocol.ModuleClassAdvertisement`). The module class advertisement uniquely identifies expected behavior and expected binding (API) to access this behavior. Each module is identified by a unique ID and an instance of the `ModuleClassID` class (`net.jxta.platform.ModuleClassID`). The purpose of the module class advertisement is to provide a description of what a particular module class ID stands for. Code running on the JXTA platform uses the module class ID of a module to designate which modules the code depends on.

The module class advertisement is not required to provide a completely formal description of the module’s behavior; it is intended for those who want to create modules with a similar functionality. It is not required to publish a module class advertisement for a module class ID to be valid, although this is a good practice.

### 6.2.2 Module Specification Advertisement

The `ModuleSpecAdvertisement` class (`net.jxta.protocol.ModuleSpecAdvertisement`) is used for specifying a module’s expected on-wire behavior and protocol implementation. The module specification advertisement associates a unique `ModuleSpecID` (`net.jxta.platform.ModuleSpecID`) to each module specification. JXTA code uses...
the module specification ID to identify a particular network-compatible family of implementations of a given module. A predefined module specification is used for designating each policy behavior of the built-in NetPeerGroup services (discovery, membership, resolver, etc.).

The module specification advertisement contains all of the necessary information to access or invoke a module. A module may be accessed through a pipe or through a proxy module code. In the case of a JXTA-enabled service (invoked via a pipe), the module specification will contain a pipe advertisement so that peers may access the service.

A module class can have multiple module specifications. For instance, an advertisement search behavior may be specified in multiple ways, using different search protocols. A module specification, which is identified by a unique module specification ID, corresponds to each possible specification. The module specification ID embeds the module class ID that points to the corresponding module class.

A module specification implies protocol compatibility between the different implementations of the specification. All implementations of a given module specification must use the same protocols and be compatible, although they may be written in different languages.

### 6.2.3 Module Implementation Advertisement

The `ModuleImplAdvertisement` class (`net.jxta.protocol.ModuleImplAdvertisement`) is used for advertising the implementation of a given module specification. There may be multiple module implementations for a given module specification. Each module implementation advertisement contains the module specification ID of the associated specification it implements. An implementation can be selected by the type of environment in which it can be used (its compatibility statement) as well as by its name, description, or the content of its parameters section.

A module implementation advertisement also provides a means to retrieve all of the data required to execute the implementation being described. This information is encapsulated in the code and URI elements of the advertisement. The interpretation of these elements is subject to the implementation of the module. For example, the standard peergroup implementation of the Java reference implementation expects the code element to contain a fully qualified Java class name that designates a subclass of the `net.jxta.platform.Module` class; it expects the URI element to contain the URI of a downloadable package (a jar file). Other execution environments could expect the code to be inlined within the code element, or to have other options.

### 6.2.4 Module Publishing and Instantiation

Modules represent applications, peergroup services, and standalone peer services. JXTA services use the module abstraction to identify the existence of a service (i.e., the module class advertisement), the specification of a service (the module specification advertisement), and an implementation of a service (the module implementation advertisement). Each advertisement can be published and discovered by other JXTA peers.

As an example of a module, consider the JXTA discovery service. It has a unique module class ID, identifying it as a discovery service — its abstract functionality. There can be multiple specifications of the discovery service, each possibly incompatible with each other. One discovery service may use different discovery policies tailored to the size of the group and its dispersion across the network, while another experiments with new policies. Each specification has a unique module specification ID that references the discovery service's module class ID. For each specification, there can be multiple implementations, each of which points to the same module specification ID.

To summarize, there can be multiple specifications of a given module class, and each may be incompatible. However, all implementations of any given specification must be compatible and communicate using the same protocol.

### 6.3 A Peer Service Example
Let’s use our RestoNet example to illustrate how the RestoPeer can publish itself as a service and how the HungryPeer can discover and use that service.

6.3.1 The RestoPeer as a Peer Service

First, we’ll modify the RestoPeer class to publish a service advertisement rather than a pipe advertisement. To do so, we’ll use the createRestoPeerService( ) method of the RestoPeer class:

```java
// This routine creates a module specification advertisement to be
// associated with a RestoPeer
private void createRestoPeerService( ) {
    try {
        // First, create the module class advertisement
        // associated with the // service. The module class advertisement advertises
        // the existence // of the service. To access the service, a peer will
        have to // discover the associated module specification
        advertisement.
        ModuleClassAdvertisement mcadv = (ModuleClassAdvertisement)
            AdvertisementFactory.newAdvertisement(
                ModuleClassAdvertisement.getAdvertisementType(  ));
        mcadv.setName("JXTAMOD:RestoNet:Service:" + brand);
        mcadv.setDescription("RestoPeer service");

        ModuleClassID mcID = IDFactory.newModuleClassID(  );
        mcadv.setModuleClassID(mcID);

        // Publish the module class advertisement in my local
        cache and to // peergroup
        disco.publish(mcadv, DiscoveryService.ADV);
        disco.remotePublish(mcadv, DiscoveryService.ADV);

        // Create the module specification advertisement
        // with the service. The module specification
        advertisement will // contain all the information necessary for a client
        to contact the // service; for example, it will contain a pipe
        advertisement that // will be used to contact the service.
        ModuleSpecAdvertisement mdadv = (ModuleSpecAdvertisement)
            AdvertisementFactory.newAdvertisement(
                ModuleSpecAdvertisement.getAdvertisementType(  ));

        // Set up some of the information fields about the
        service. In this
```
// example, we set the name, provider, and version, along with a pipe advertisement. The module creates an input pipe to listen on
// this pipe’s endpoint.
mdadv.setName("JXTASPEC:RestoNet:Service:" + brand);
mdadv.setVersion("Version 1.0");
mdadv.setCreator("sun.com");
mdadv.setModuleSpecID(IDFactory.newModuleSpecID(mcID));
mdadv.setSpecURI("http://www.jxta.org/tutorial/RestoPeer.jar");

// Set a pipe advertisement for the service. The pipe is used to communicate with the service. The HungryPeer client MUST use this pipe to talk to the service. The pipe advertisement is stored in the service advertisement parameters; the client retrieves it from the parameters.
mdadv.setPipeAdvertisement(myAdv);

// Display the advertisement as a plain-text document
StructuredTextDocument doc = (StructuredTextDocument) mdadv.getDocument(new MimeMediaType("text/plain"));
StringWriter out = new StringWriter();
doc.sendToWriter(out);
System.out.println(out.toString());
out.close();

// Publish the service advertisement
disco.publish(mdadv, DiscoveryService.ADV);
disco.remotePublish(mdadv, DiscoveryService.ADV); catch (Exception ex) {
System.out.println("Error publishing RestoPeer Advertisement");
}

The createRestoPeerService() method creates a module class advertisement and a module specification advertisement using the same technique we used in Chapter 4. After creating each type of advertisement, we must populate each of its attributes by calling the appropriate setter methods.

There is also one new feature in this service advertisement. JXTA-enabled services can publish a pipe advertisement to be used by applications to connect to and invoke the service. The setPipeAdvertisement() method sets this pipe advertisement; in our example, we use the RestoPeer pipe advertisement (myAdv) created by the createRestoPipe() method. This is the same pipe advertisement we published in Chapter 5.

This completes the initialization of the service advertisement. Next, we print out the advertisement (using our standard procedure of generating a structured XML document); this is just for our own information. Finally, the service advertisement is published in the peergroup: first in the local peer cache via the publish() method, and then in the known peergroup rendezvous via the remotePublish() method.
This method requires two slight changes to the rest of the RestoPeer class. First, we must invoke this method in the `startJxta()` method:

```java
// Method to start the JXTA platform, join the RestoNet peergroup, and advertise the RestoPeer service
private void startJxta() {
    try {
        // Discover and join (or start) the default peergroup
        netpg = PeerGroupFactory.newNetPeerGroup();
    } catch (PeerGroupException e) {
        // Couldn’t initialize; can’t continue
        System.out.println("Fatal error : creating the NetPeerGroup");
        System.exit(1);
    }

    // Discover (or create) and join the RestoNet peergroup
    try {
        joinRestoNet();
    } catch (Exception e) {
        System.out.println("Can’t join or create RestoNet");
        System.exit(1);
    }

    // Discover (or create) and publish a RestoPeer pipe to receive auction requests for fries from HungryPeers
    if (!createRestoPipe()) {
        System.out.println("ABorting due to failure to create RestoPeer pipe");
        System.exit(1);
    }

    // Advertise the RestoPeerService
    createRestoPeerService();

    // Start the RestoPeer server loop to respond to HungryPeer’s fries requests
    handleFriesRequest();
}
```

Second, we no longer need to publish the pipe advertisement within the `createRestoPipe()` advertisement:

```java
private boolean createRestoPipe() {
    ..... // We have the advertisement, so we can publish our pipe
    // advertisement into our local cache
    disco.publish(myAdv, Discovery.ADV);
    /* disco.remotePublish(myAdv, Discovery.ADV); */
    .....}
```
Note that we still publish the advertisement to the local cache; the next time this peer starts, we want it to find and use the same pipe advertisement. But we don’t publish the advertisement to the peergroup, since we want peers to discover the peer through the service advertisement.

6.3.2 The HungryPeer Example

Let’s look now at the corresponding version of HungryPeer. Previously, the HungryPeer class looked for a pipe advertisement to find RestoPeers; the modified version of HungryPeer uses a new `discoverRestoServices()` method that first discovers RestoPeer service advertisements, and then extracts the service pipe advertisements:

```java
// Discover RestoPeers that have joined RestoNet
// We discover RestoPeers via the module specification advertisement that
// was published to advertise a RestoPeerService
private void discoverRestoServices() {
    int found = 0;                     // Number of RestoPeers
    int count = 10;                    // Discovery retries
    ModuleSpecAdvertisement restoSrv;  // RestoPeer service advertisement

    System.out.println(
        "Locating RestoPeers in the RestoNet Peergroup";

    // Try to find at least two RestoPeers (an arbitrary number)
    while (count-- > 0) {
        try {
            // Check if we already have restaurant advertisements
            Enumeration ae =
                disco.getLocalAdvertisements(DiscoveryService.ADV, "Name",
                "JXTASPEC:RestoNet:Service:*");

            // If we found some advertisements in the cache,
            // add them to the list
            if (ae != null && ae.hasMoreElements( )) {
                // Reset count and RestoPeer advertisements,
                // as we have
                // just retrieved all advertisements,
                found = 0;
                restoPeerAdvs.removeAllElements( );
                while (ae.hasMoreElements( )) {
                    restoSrv =
                        (ModuleSpecAdvertisement)
                        ae.nextElement( );

                    // Extract the pipe advertisement from the
                    service
                    // module specification advertisement. The pipe
```
// advertisement is used to connect to the service; it is part of the parameter section of the advertisement.

restoSrv.getPipeAdvertisement();
++found;
}
if (found > 1)
    break;          // Want to find at least two

// Did not find enough advertisements in the cache
// Send a remote discovery request to search for more
// RestoPeer service advertisements
disco.getRemoteAdvertisements(null, DiscoveryService.ADV, "Name", "JXTASPEC:RestoNet:Service:*", 5, null);

// Give the peers a chance to respond
try {
    Thread.sleep(timeout);
} catch (InterruptedException e) {} 
} catch (IOException e){
    // Found nothing! Move on.
}

// Completed RestoPeer discovery
System.out.println("Found " + found + " RestoPeer Service(s)");

We invoke this method instead of invoking the discoverRestoPeers( ) method we used in our last example. It uses essentially the same procedure to discover RestoPeers, but it has two main differences. First, the name of the advertisement that we're looking for has changed to "JXTASPEC:RestoNet:Service:*". The second change is in how we handle the discovered advertisements. In this case, the advertisements that we discovered are service advertisements; in order to obtain the pipe advertisement needed to talk to the service, we must use the getPipeAdvertisement( ) method. We store this pipe advertisement in the restoPeerAdvs vector, just as we did before.

### 6.3.3 Running the Example

As usual, we must set up separate directories from which to run each peer. When they run, they will behave the same way as the peers did in the last example: the RestoPeer will find or create the RestoNet peergroup and advertise its service, the HungryPeer will find the RestoNet peergroup and locate the RestoPeers within it, the HungryPeer will send a request to the RestoPeer, and the RestoPeer will send a response. Because we print out the service advertisement within the RestoPeer, that peer will give us some new output:

Attempting to Discover the RestoNet PeerGroup
Could not find the RestoNet peergroup; creating one
RestoNet Restaurant (Chez JXTA) is on-line
Attempting to Discover the Restaurant RestoPipe
Created the Restaurant Pipe Advertisement

jxta:MSA :
  MSID : urn:jxta:uuid-A0783B698094493295ED92850978D367B755B095
86F04473B5308CDD157168F806
  Name : JXTASPEC:RestoNet:Service:Chez JXTA
  Crtr : sun.com
  SURI : http://www.jxta.org/tutorial/RestoPeer.jar
  Vers : Version 1.0
  jxta:PipeAdvertisement(xmlns:jxta="http://jxta.org" ) :
    Id : urn:jxta:uuid-EC10CBA8152A499AA7E2AAEEBE37077351
1A69B60B5B4894B92CA33ABD2C24B604
    Type : JxtaUnicast
    Name : RestoNet:RestoPipe:Chez JXTA

RestoNet Restaurant (Chez JXTA) waiting for HungryPeer requests

Here we can see the pipe advertisement embedded within the service advertisement.

6.3.4 Pipe Versus Service Advertisement Publishing

We have demonstrated two ways to publish a service. In Chapter 5, the RestoPeer service was advertised via a pipe advertisement. In this chapter, a service advertisement is used to publish the service. There are some key differences between these two methods. Peers should advertise their services via a pipe advertisement when there is no need for other peergroup members to create a new instance of the service. By publishing only a pipe advertisement, other members of the peergroup can connect to the service only by creating a pipe; they do not have the necessary information to instantiate a new instance of the service. Only the peer running the service knows how to instantiate the service.

On the other hand, publishing the service advertisement allows any peer to discover it and create a new instance of it. Every peer instantiating the service can publish its own copy of the service advertisement. More peers can run the service, which makes the service more available. This allows the service to be replicated dynamically within the peergroup, which increases the overall reliability of a service.

6.4 A Peergroup Service Example

Peergroup services are advertised as part of the peergroup advertisement. Therefore, its service is published when its peergroup advertisement is published. Each member of the peergroup has access to the peergroup advertisement and will typically instantiate each peergroup service.

An implementation of a peergroup may define its own policy to determine how peergroup services are instantiated. The v1.0 platform automatically instantiates every peergroup service listed in the peergroup advertisement. However, this is only one possible policy. Other implementations may use different policies, such as loading on-demand (i.e., instantiating a service only if somebody is trying to use it), or letting the peer decide which peergroup services to instantiate. The latter policy is useful when a memory-constrained device tries to join a peergroup. Since the device may not have enough memory to run all peergroup services, it can instead run only the necessary services. Publishing a service as part of the peergroup advertisement means that every member of the peergroup knows about the service without any further discovery: there is no need to discover the
service since the service advertisement is present in the peergroup advertisement. When the
peergroup advertisement is found, the peer can extract all of the necessary information about a
desired service from the peergroup advertisement. Peer services, on the other hand, require two
discovery requests: one to discover the peergroup and one to discover the service.
Peergroup services are closely related to their associated peergroup. It is not possible to add or
remove a peergroup service from a peergroup advertisement after the peergroup advertisement has
been published. Therefore, peergroup services are determined at the creation of the peergroup.
This is a key difference between peer and peergroup services. A peer service can be published at
any time, whereas a peergroup service is published only when the peergroup advertisement is
published.\footnote{A peergroup advertisement is republished each time a new peer joins a peergroup.
Each republished advertisement contains some information about the new peer (e.g.,
the peer ID), but the service advertisements cannot change.}

6.4.1 The RestoPeer Peergroup Service

We’ll start our example by converting the RestoPeer peer service to a peergroup service. This is a
two-step process. First, we convert the RestoPeer class to advertise the RestoNet service within
the peergroup. When the RestoPeer creates the RestoNet peergroup, it will include a service
advertisement for the auction service. However, the implementation of that service will be moved
to a new class: the \texttt{RestoNetService} class.

First, here’s the new RestoPeer class:

```java
import java.io.*;
import java.util.*;
import java.net.URL;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.exception.PeerGroupException;
import net.jxta.discovery.DiscoveryService;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.Advertisement;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentUtils;
import net.jxta.pipe.PipeService;
import net.jxta.pipe.PipeID;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;
import net.jxta.protocol.ModuleSpecAdvertisement;
import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.id.IDFactory;
import net.jxta.id.ID;
import net.jxta.platform.ModuleClassID;
import net.jxta.platform.ModuleSpecID;
import net.jxta.impl.peergroup.StdPeerGroupParamAdv;

// RestoPeer represents a restaurant that receives auction
requests for
```
// French fries from HungryPeers. RestoPeers offers three sizes of French fries (small, medium, large). Each restaurant assigns a different price for each size and offers a special. Each restaurant is uniquely identified by its brand name.

public class RestoPeer {

    private PeerGroup netpg = null; // The NetPeerGroup
    private PeerGroup restoNet = null; // The RestoNet peergroup

    private String brand = "Chez JXTA"; // Brand of this restaurant

    // Services within the RestoNet peergroup
    private DiscoveryService disco = null; // Discovery service
    private PipeService pipes = null; // Pipe service
    private PipeAdvertisement myAdv = null; // My RestoPeer pipe

    private int timeout = 3000; // Discovery wait time-out

    // IDs within RestoNet
    private ModuleClassID mcID = IDFactory.newModuleClassID();
    private ModuleSpecID msID = IDFactory.newModuleSpecID(mcID);
    private static PeerGroupID restoPeerGroupID;

    // Main method to start our RestoPeer
    public static void main(String args[]) {
        RestoPeer myapp = new RestoPeer();
        myapp.startJxta();
        System.exit(0);
    }

    // Method to start the JXTA platform, join the RestoNet peergroup, and advertise the RestoPeer service
    private void startJxta() {
        try {
            // Discover and join (or start) the default peergroup
            netpg = PeerGroupFactory.newNetPeerGroup();
            } catch (PeerGroupException e) {
                System.out.println("Fatal error : creating the NetPeerGroup");
                System.exit(1);
            }
try {
    joinRestoNet(  );
} catch (Exception e) {
    System.out.println("Can’t join or create RestoNet");
    System.exit(1);
}

synchronized(RestoPeer.class) {
    try {
        RestoPeer.class.wait(  );
    } catch (InterruptedException ie) {
        System.out.println("Interrupted; exiting");
    }
}

private void joinRestoNet(  ) throws Exception {
    int count = 3; // Maximum number of attempts to discover
    System.out.println("Attempting to Discover the RestoNet PeerGroup");

    // Get the discovery service from the NetPeerGroup
    DiscoveryService hdisco = netpg.getDiscoveryService( );

    Enumeration ae = null; // Holds the discovered peers

    // Loop until we discover the RestoNet or until we’ve exhausted the
    // desired number of attempts
    while (count-- > 0) {
        try {
            // Search first in the local peer cache to find the RestoNet
            // peergroup advertisement
            ae = hdisco.getLocalAdvertisements(DiscoveryService.GROUP, "Name", "RestoNet");

            // If we found the RestoNet advertisement, we are done
            if ((ae != null) && ae.hasMoreElements(  ))
                break;

            // If we did not find it, we send a discovery request
            hdisco.getRemoteAdvertisements(null, DiscoveryService.GROUP, "Name", "RestoNet");
        }
    }
}
DiscoveryService.GROUP, "Name", "RestoNet", 1, null);

    // Sleep to allow time for peers to respond to the
    // discovery request
    try {
        Thread.sleep(timeout);
    } catch (InterruptedException ie) {}  
    } catch (IOException e) {
        // Found nothing! Move on.
    }

PeerGroupAdvertisement restoNetAdv = null;

    // Check if we found the RestoNet advertisement. If we
didn’t, then
    // either we are the first peer to join, or no other
RestoNet peers
    // are up. In either case, we must create the RestoNet
peergroup.

    if (ae == null || !ae.hasMoreElements()) {
        System.out.println(
            "Could not find the RestoNet peergroup;
creating one");
        try {
            // Create the RestoNet peergroup
            restoNet = createRestoPeerGroup();

            // Get the peergroup advertisement
            restoNetAdv = netpg.getPeerGroupAdvertisement();
        } catch (Exception e) {
            System.out.println(
                "Error in creating RestoNet PeerGroup");
            throw e;
        }
    } else {
        // The RestoNet advertisement was found in the
        // cache; this means we can join the existing RestoNet
        peergroup
        try {
            restoNetAdv = (PeerGroupAdvertisement) ae.nextElement();
            restoNet = netpg.newGroup(restoNetAdv);
            System.out.println(
                "Found the RestoNet Peergroup advertisement");
        } catch (Exception e) {
            System.out.println(
                "Error in creating RestoNet PeerGroup from
existing adv");
try {
    // Get the discovery and pipe services for the
    // peer group
    disco = restoNet.getDiscoveryService();
    pipes = restoNet.getPipeService();
} catch (Exception e) {
    System.out.println("Error getting services from
    RestoNet");
    throw e;
}

System.out.println("RestoNet Restaurant (" + brand + ") is on-line");
return;

// This method is used to create a new instance of the
// peer group. Peer groups are implemented as modules, which
// are used in
// JXTA to load and manage dynamic code on a peer. A
// peer group is
// represented by a set of advertisements:
// 1) A peer group advertisement that advertises the peer group
// 2) A module spec advertisement that uniquely specifies
// the peer group
// (set of peer group services)
// 3) A module implementation advertisement that describes a peer group
// implementation
// This method must create all these advertisements.
private PeerGroup createRestoPeerGroup() {

    // Use a unique peer group ID as a constant so that the same
    // peer group ID is used each time the RestoNet peer group is
    // created. It is essential that each RestoPeer use
    // the same
    // unique ID so that two peers do not create different
    // IDs for
    // the RestoNet peer group.
    // The UUID in the URL constructor was generated via
    // the
    // shell mkpgrp command that created a new peer group
    // with a
    // unique peer group ID.
    try {
        restoPeerGroupID = (PeerGroupID) IDFactory.fromURL(

// Create a new module implementation advertisement that will represent the new RestoPeer peergroup service. // Assign a new specification ID that uniquely identifies the RestoPeer peergroup service. This specification ID must be shared with all instances of the RestoNet peergroup; it is also created via the mkpgrp shell command.

ModuleSpecID msrvID = null;
try {
    msrvID = (ModuleSpecID) IDFactory.fromURL(new URL("urn","",
"jxta:uuid-
4d6172676572696e204272756f2002"));
} catch (java.net.MalformedURLException e) {
    System.err.println("Can't create restoPeerGroupID: " + e);
    System.exit(1);
} catch (java.net.UnknownServiceException e) {
    System.err.println("Can't create restoPeerGroupID: " + e);
    System.exit(1);
}

// Create a pipe advertisement to be used by HungryPeers to communicate with the RestoNet peergroup service.
PipeAdvertisement myAdv = (PipeAdvertisement)
AdvertisementFactory.newAdvertisement(PipeAdvertisement.getAdvertisementType( ));

// Again we assign a hardwired ID to the pipe, so every time a peer recreates the RestoNet peergroup, the same pipe ID is used
try {
    myAdv.setPipeID((PipeID) IDFactory.fromURL(new URL("urn","",
"jxta:uuid-" +
try {
    // Create the module implementation advertisement that represents the RestoPeer service.
    // Peergroup services are also represented by modules since a peergroup service needs to be dynamically loaded when the corresponding peergroup is instantiated on the peer.
    // The constructor of the service module implementation takes a set of arguments to describe the service:
    // msrvID: unique specification ID of the RestoNet service
    // "RestoPeerService": the java class implementing the service
    // "http://jxta.org/tutorial/RestoPeer.jar": where to find jar file
    // "sun.com": provider of the service
    // "RestoPeer Service Module Implementation": a description of the service

    // Create a new peergroup that is a clone of the NetPeergroup but with the added RestoPeerService
    return createPeerGroup(netpg, "RestoNet", restoImplAdv);
} catch (java.net.MalformedURLException e) {
    System.err.println("Can’t create restoPeer PipeID: " + e);
    System.exit(1);
} catch (java.net.UnknownServiceException e) {
    System.err.println(" Can’t create restoPeer PipeID: " + e);
    System.exit(1);
}
// This is a utility method to create a new peergroup implementation.
// The new peergroup will have the same services as the parent peergroup
// and an additional service (the srvImpl).
// The method creates the new Peergroup module implementation
// advertisement and publishes it, creates the associated peergroup
// advertisement and publishes it, and instantiates the peergroup using
// the new peergroup advertisement.
//     parent: parent peergroup the new group is created in
//     groupName: name of the new group
//     srvImpl: new peergroup service to add to the default set of NetPeerGroup services
private PeerGroup createPeerGroup(PeerGroup parent, String groupName, ModuleImplAdvertisement srvImpl) {
    PeerGroup myPeerGroup = null;
    PeerGroupAdvertisement myPeerGroupAdvertisement = null;
    ModuleImplAdvertisement myModuleImplAdv = null;
    // Create the peergroup module implementation adding the new peergroup service
    myModuleImplAdv = createPeerGroupModuleImplAdv(parent, srvImpl);
    // Publish the new peergroup module implementation in the parent peergroup
    DiscoveryService parentDiscovery = parent.getDiscoveryService();
    // Publish advertisement in the parent peergroup with the default lifetime and expiration time for remote publishing
    try {
        parentDiscovery.publish(myModuleImplAdv, DiscoveryService.ADV, PeerGroup.DEFAULT_LIFETIME, PeerGroup.DEFAULT_EXPIRATION);
        parentDiscovery.remotePublish(myModuleImplAdv, DiscoveryService.ADV, PeerGroup.DEFAULT_EXPIRATION);
    } catch (java.io.IOException e) {
        System.err.println("Cannot publish the new peergroup ModuleImplAdv");
        return null;
    }
}
System.out.println(
  "Published new PeerGroup ModuleImpl advertisement for : "
  + groupName);

// Create and publish the new peer group advertisement
// corresponding to the new peer group implementation
myPeerGroupAdvertisement =
createPeerGroupAdvertisement(
  myModuleImplAdv, groupName);

// Publish the advertisement in the parent peer group
// with the
// default lifetime and expiration time
try {
  parentDiscovery.publish(myPeerGroupAdvertisement,
    DiscoveryService.GROUP,
    PeerGroup.DEFAULT_LIFETIME,
    PeerGroup.DEFAULT_EXPIRATION);
  parentDiscovery.remotePublish(myPeerGroupAdvertisement,
    DiscoveryService.GROUP,
    PeerGroup.DEFAULT_EXPIRATION);
} catch (java.io.IOException e) {
  System.err.println(
    "Cannot create the new PeerGroup Advertisement");
  return null;
}

System.out.println(
  "Published the new PeerGroup advertisement for : " +
  groupName);

try {
  // Finally, instantiate the new peer group
  myPeerGroup =
  parent.newGroup(myPeerGroupAdvertisement);
  System.out.println(
    "Instantiated the new PeerGroup " +
    groupName);
} catch (net.jxta.exception.PeerGroupException e) {
  System.err.println("Cannot create the new Peer
  Group");
  return null;
}

return myPeerGroup;

// This method is a generic method used to create a new
// module implementation advertisement for representing a
// peer group
// service. Each module implementation corresponds to a
// unique
// module specification ID that uniquely identifies a
peergroup service.
// A peergroup service is represented by:
//     ID: unique module specification ID identifying the
new peergroup
//     service
//     code: Java class that implements the peergroup
service
//     uri: location where we can download the Java class
implementing
//     the service
//     provider: identity of the provider
//     desc: description of the service
//     padv: pipe advertisement to invoke the service
private  ModuleImplAdvertisement
createServiceModuleImplAdv(
    ModuleSpecID id,
    String       code,
    String       uri,
    String       provider,
    String       desc,
    PipeAdvertisement padv) {

    // Construct a new module implementation advertisement
    ModuleImplAdvertisement ServiceModuleImplAdv =
        (ModuleImplAdvertisement)
        AdvertisementFactory.newAdvertisement(
            ModuleImplAdvertisement.getAdvertisementType(  
            ModuleImplAdvertisement));

    // Set a unique module specification ID associated
with this
    // service implementation
    ServiceModuleImplAdv.setModuleSpecID(id);

    // Set the java class that implements this service
    ServiceModuleImplAdv.setCode(code);

    // Set the description of the new service
    ServiceModuleImplAdv.setDescription(desc);

    // Define the compatibility of Java JDK 1.4 for
loading this module
    StructuredTextDocument doc = (StructuredTextDocument)
        StructuredDocumentFactory.newStructuredDocument(
            new MimeMediaType("text", "xml"), "Comp")
            Element e = doc.createElement("Fmt", "JDK1.4")
            doc.appendChild(e);
            e = doc.createElement("Bind", "V1.0 Ref Impl")
            doc.appendChild(e);
            ServiceModuleImplAdv.setCompat(doc);
// Set the URL where we can download the code for this module
ServiceModuleImplAdv.setUri(uri);

// Set the provider for this module
ServiceModuleImplAdv.setProvider(provider);

// Set the pipe advertisement into a parameter document to be used
// by a peer to connect to the service
StructuredTextDocument paramDoc =
(StructuredTextDocument)
StructuredDocumentFactory.newStructuredDocument
(StructuredDocumentUtil.copyElements(paramDoc,
paramDoc,
(Element) padv.getDocument(new
MimeMediaType("text/xml")));
ServiceModuleImplAdv.setParam(paramDoc);

// Return the new module implementation for this service
return ServiceModuleImplAdv;

// Create a module implementation advertisement that represents a
// peergroup that is a clone of the NetPeerGroup, but adds
// a new peergroup service to the peergroup
private ModuleImplAdvertisement createPeerGroupModuleImplAdv(
    PeerGroup parent,
    ModuleImplAdvertisement serviceModuleImplAdv)
{
    ModuleImplAdvertisement allPurposePeerGroupImplAdv = null;
    StdPeerGroupParamAdv PeerGroupParamAdv = null;
    try {
        // Clone the parent (NetPeerGroup) implementation to get
        // a module implementation
        allPurposePeerGroupImplAdv = parent.getAllPurposePeerGroupImplAdvertisement(  );
    }
    catch (Exception e) {
        System.err.println("Cannot get allPurposePeerGroupImplAdv " + e);
        return null;
    }
Get the parameter field that contains all the peergroup services associated with the peergroup

```java
try {
    PeerGroupParamAdv = new StdPeerGroupParamAdv(allPurposePeerGroupImplAdv.getParam());
} catch (PeerGroupException e) {
    System.err.println("Cannot get StdPeerGroupParamAdv " + e);
    return null;
}
```

Get the hashtable containing the list of all the peergroup services from the parameter advertisements

```java
Hashtable allPurposePeerGroupServicesHashtable = PeerGroupParamAdv.getServices();
```

Add our new peergroup service to the list of peergroup services

```java
allPurposePeerGroupServicesHashtable.put(mcID, serviceModuleImplAdv);
```

Update the PeerGroupModuleImplAdv with our new list of peergroup services

```java
allPurposePeerGroupImplAdv.setParam((Element) PeerGroupParamAdv.getDocument(new MimeMediaType("text/xml")));
```

Set the new unique specification ID that identifies this new peergroup implementation

```java
allPurposePeerGroupImplAdv.setModuleSpecID(msID);
```

We are done creating our new peergroup implementation

```java
return allPurposePeerGroupImplAdv;
```

This utility method is used to create a peergroup advertisement associated with a module peergroup implementation

```java
private PeerGroupAdvertisement createPeerGroupAdvertisement(
    ModuleImplAdvertisement implAdv, String groupName) {

    // Create a new peergroup advertisement
    PeerGroupAdvertisement myadv = (PeerGroupAdvertisement)
        AdvertisementFactory.newAdvertisement(
            PeerGroupAdvertisement.getAdvertisementType(
```
Instead of creating a new group ID we use the predefined peergroup ID, so we create the same peergroup ID each time the group is created:

```java
myadv.setPeerGroupID(restoPeerGroupID);
```

Assign the predefined module specification ID that corresponds to our peergroup module implementation:

```java
myadv.setModuleSpecID(implAdv.getModuleSpecID());
```

Assign peergroup name:

```java
myadv.setName(groupName);
```

Set the peergroup description:

```java
myadv.setDescription("This is RestoNet Peergroup");
```

return myadv;

There are significant differences in this implementation of the RestoPeer class. The `startApp()` method begins as it always has: it finds (or creates) the default peergroup and uses that group to join the RestoNet peergroup. After this, however, the main thread simply waits: it no longer creates any pipes, nor does it call the `handleFriesRequest()` method. What’s happened here is that the new implementation of the `joinRestoNet()` method (which we’ll discuss in more detail in the next section) has included the RestoNet service as part of the peergroup advertisement. Information about the RestoNet service in the peergroup advertisement includes the name of a class (the `RestoPeerService` class). When a HungryPeer uses the service represented by the new RestoNet advertisement, it will dynamically instantiate the `RestoPeerService` class.

This concept is familiar to us from how other peergroup services work. The discovery service, for example, is handled in just this manner: when a peer discovers a peergroup (including the `NetPeerGroup`), the discovery service associated with that peergroup is automatically instantiated within the peer.

This means that there are significant differences in how the peergroup advertisement is created in the `joinRestoNet()` method. Instead of creating a `ModuleSpecAdvertisement` object as we’ve done before, we now call the new `createRestoPeerGroup()` method. This method (and its helper methods) is responsible for creating the peergroup advertisement that includes the RestoNet service.

### 6.4.1.1 The pipe ID

The RestoPeer service uses a hardwired pipe ID to ensure that all implementations of the service use the same pipe ID; this is done by constructing the pipe ID from a URL. In general, all JXTA IDs (not just pipe IDs) are represented by URLs with a protocol of "jxta" (e.g., `jxta://...`). In order to create a pipe ID, you must generate the appropriate string from which the URL is constructed. The easiest way to do this is by using the following application:

```java
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.pipe.PipeID;
import net.jxta.id>IDFactory;

public class GenerateURL {
```
public static void main(String[] args) throws Exception {
    PeerGroup pg = PeerGroupFactory.newNetPeerGroup();
    PipeID ID = IDFactory.newPipeID(pg.getPeerGroupID());
    System.out.println(id);
}
}

Run this code like any other JXTA application: set it up in a separate directory and configure it using the configurator tool. The pipe ID is written as a URL; you can paste the output from this application into your own service when you need a URL from which to create a unique pipe ID.

6.4.1.2 The separated service class

Peergroup services should be packaged into separate class files. The RestoPeer class that we've just seen is essentially a bootstrapping class: it ensures that the correct service is included within the peergroup advertisement. But it doesn't provide the implementation of that service; in this case we expect that each peer who needs the service will instantiate it. These peers shouldn't instantiate the class that creates the advertisement; they need to instantiate the class that provides the logic of the service. This logic is placed into the RestoPeerService class: the class set as the main class within the service advertisement.

The RestoPeerService class implements the logic of the service as follows:

```java
import java.io.*;
import java.util.*;
import net.jxta.peergroup.PeerGroup;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.Advertisement;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.discovery.Discovery;
import net.jxta.pipe.Pipe;
import net.jxta.pipe.PipeID;
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.OutputPipe;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.protocol.ServiceAdvertisement;
import net.jxta.service.Service;
import net.jxta.endpoint.Message;

// The RestoPeer class is an example of a peergroup service
// This service is registered as a RestoNet peergroup service
public class RestoPeerService implements Service {

    private String brand = "Chez JXTA";         // Brand of restaurant
    private String specials = "large ($3.00)";  // Restaurant special
    private PeerGroup restoNet = null;          // RestoNet peergroup
    private Pipe pipes = null;                  // Pipe service in RestoNet
```
private ServiceAdvertisement srvadv = null; // Service advertisement
private PipeAdvertisement myAdv = null; // RestoNet pipe advertisement
private InputPipe pipeIn = null; // Pipe that we listen to
private int rtimeout = 8000; // Resolver time-out
private int timeout = 2000; // Discovery time-out
private Discovery disco; // Discovery service

// Service objects are not manipulated directly to protect the usage
// of the service. A service interface is returned to access the
// service.
public Service getInterface( ) {
    return this;
}

// Returns the advertisement for this service.
public Advertisement getAdvertisement( ) {
    return srvadv;
}

// The required initiation method of the service
public void init(PeerGroup group, Advertisement adv) {

    // Save the RestoNet pointer, the pipe, and the RestoPeer service
    restoNet = group;
pipes = restoNet.getPipe( );
srvadv = (ServiceAdvertisement) adv;
    System.out.println("Initialization Special Service: "+
            srvadv.getName( ));

    // Extract the pipe advertisement from the service advertisement
    myAdv = srvadv.getPipe( );

    // Create the input pipe to receive incoming request
    if (!createRestoPipe( )) {
        System.out.println("Abort due to failure to create RestoPeer pipe");
    }

    // Start the RestoPeer service loop thread to respond to
// HungryPeer’s fries requests. Start a new thread for processing
// all the requests.
HandleFriesRequestThread peerRequest =
    new HandleFriesRequestThread(  );
peerRequest.start(  );
}

// The required startApp method of the JXTA Application interface
public int startApp (String args[]) {
    // nothing to do here
    return 0;
}

// The required stopApp method of the JXTA Application interface
public void stopApp(  ) {
}

// Create the pipe associated with this RestoPeer. First, discover if a
// pipe advertisement exists; if it doesn’t, create and publish it.
private boolean createRestoPipe(  ) {
    ... same as previous implementation ... 
}

// Thread to handle fries auction requests from HungryPeers.
// The method waits for HungryPeer requests’ pipe messages to arrive.
// Incoming requests contain a pipe advertisement to respond to the
// HungryPeers requester and a fries size.
private class HandleFriesRequestThread extends Thread {

    // Start the request-processing thread
    public void run(  ) {
        System.out.println("RestoNet Restaurant (" + brand + 
              ") waiting for HungryPeer requests");

        // Loop waiting for HungryPeer requests
        while (true) {
            handleFriesRequest(  );
        }
    }

    private void handleFriesRequest(  ) {
        ... same as previous implementation ... 
    }
}
The RestoPeerService class is an example of a JXTA-enabled service. The service uses an input pipe to listen for incoming requests. It processes the requests by reading a message from the input pipe and opening an output pipe to the requester to send the response.

The RestoPeerService class implements the Service interface (net.jxta.service.Service), which in turn implements the Application (net.jxta.platform.Application) interface. From a programming perspective, service modules look like a Java applet or servlet: the underlying container (the JXTA platform, in this case) instantiates them and controls their lifecycle through a series of callback methods. The RestoPeerService class is responsible for implementing each of these callbacks.

The init() method is called just after the service has been instantiated. The peergroup in which the service is running is passed to this method so that the service can find all of the necessary core services (as well as its advertisement, which it must save so that it can return from the getAdvertisement() method, another method of the Service interface).

The startApp() method is called just before the service officially starts. The service can finish any necessary initialization here. The stopApp() method is called just before the service is stopped (though there is no assurance that this method will be called; the program could be terminated before the platform can call this method). This allows the service to clean up any resources before it exists.

The actual interface that the service implements is returned by the getInterface() method, which is called by the platform when a peer is going to interact with the service. We return this in the example; you could also return a clone of the service if you feel it is more appropriate.

Note that the init() method starts a thread to handle any incoming requests. A JXTA-enabled service is responsible for managing its own threads. When the platform calls the init() method of a service, it expects the method to return after initialization is complete. Therefore, the init() method creates and starts a new thread — the HandleFriesRequestThread class — to handle and process incoming requests.

The run() method of the HandleFriesRequestThread class is identical to the handleFriesRequest() method discussed earlier: the method reads a message from the service pipe, extracts the request document, processes the request, and constructs a bid message that is sent by opening a pipe to the requester.

When we run the new RestoPeer class, we get this output:

```
Attempting to Discover the RestoNet PeerGroup
Could not find the RestoNet peergroup create one
RestoNet Restaurant (Chez JXTA) waiting for HungryPeer requests
Notice that the waiting for HungryPeer requests message is printed before
the RestoPeer on-line message, indicating that the RestoPeer service instantiation was completed before the peer finished joining RestoNet. The service initialization is performed as part of the peergroup instantiation.
```

If you modify the RestoNet class to print the RestoNet peergroup advertisement by extracting the advertisement document (using the getDocument() method), you'll get the following output:

```
Services :
    jxta:ServiceAdvertisement :
        Name : jxta.service.discovery
        Version : JXTA1.0
        Keywords : discovery
        Uri : http://www.jxta.org/download/jxta.jar
        Provider : sun.com
        Security : TBD
        Code : net.jxta.impl.discovery.DiscoveryService
    jxta:ServiceAdvertisement :
```
Name : jxta.service.pipe
Version : JXTA1.0
Keywords : pipe
Uri : http://www.jxta.org/download/jxta.jar
Provider : sun.com
Security : TBD
Code : net.jxta.impl.pipe.PipeService

jxta:ServiceAdvertisement :
Name : RestoNet:Service:Chez JXTA
Version : Version 1.0
PipeService :
   jxta:PipeAdvertisement :
      ID : jxta://A2CD5C0D41A146799F4DA73AC4EE9F7481F0BCCB5
      name : RestoNet:RestoPipe:Chez JXTA
      Uri : http://www.jxta.org/tutorial/RestoPeer.jar
      Provider : jxta.org
      Security : None
      Code : RestoPeerService

jxta:ServiceAdvertisement :
Name : jxta.service.resolver
Version : JXTA1.0
Keywords : resolver
Uri : http://www.jxta.org/download/jxta.jar
Provider : sun.com
Security : TBD
Code : net.jxta.impl.resolver.ResolverService

jxta:ServiceAdvertisement :
Name : jxta.service.rendezvous
Version : JXTA1.0
Keywords : rendezvous
Uri : http://www.jxta.org/download/jxta.jar
Provider : sun.com
Security : TBD
Code : net.jxta.impl.rendezvous.RendezVousImpl

jxta:ServiceAdvertisement :
Name : jxta.service.peerinfo
Version : JXTA1.0
Keywords : peerinfo
Uri : http://www.jxta.org/download/jxta.jar
Provider : sun.com
Security : TBD
Code : net.jxta.impl.peerPeerInfoService

jxta:ServiceAdvertisement :
Name : jxta.service.membership
Version : JXTA1.0
Keywords : Membership
Uri : http://www.jxta.org/download/jxta.jar
Provider : sun.com
Security : TBD
Code : net.jxta.impl.membership.NullMembership
6.4.2 The HungryPeer Implementation

Now we'll look at the HungryPeer and see how it can operate within the peergroup service. The only modification required is to replace the `discoverRestoPeers()` method, which is used to send a discovery request to search for RestoPeer service advertisements, with the `discoverRestoServices()` method. Now we find the service advertisement as part of the peergroup advertisement:

```java
private void discoverRestoServices() {
    int found = 0;                  // Number of RestoPeers
    ServiceAdvertisement restoSrv;  // RestoPeer service advertisement
    System.out.println(            
        "Locating RestoPeers in the RestoNet Peergroup");

    try {                           
        // Extract the RestoPeer service from the RestoNet peergroup advertisement since it is now part of the RestoNet peergroup
        restoSrv = (ServiceAdvertisement) 
            restoNet.lookupService("RestoNet.RestoPeer.Service").getAdvertisement(  );

        // Extract the pipe advertisement for the RestoPeer service
        restoPeerAdvs.addElement(restoSrv.getPipe(  ));
        ++found;
    } catch (ServiceNotFoundException ex) {
    }
    // Completed RestoPeer discovery
    System.out.println("Found " + found + " RestoPeers Service");
}
```

The `discoverRestoServices()` method shows how to extract the RestoPeer service from the RestoNet peergroup advertisement. The `lookupService()` method is used to look for a specific service. The `lookupService()` method takes as an argument the name of a service ("RestoNet.RestoPeer.Service"); it returns a `Service` object. The `getAdvertisement()` method is used to extract the corresponding service advertisement, from which the pipe advertisement is extracted via the `getPipe()` method.

The `discoverRestoServices()` method is trivial because all instances of the RestoPeer service use the same pipe ID: we need to find only one service instance to get the pipe ID. Some peergroup services may allow each instance to have a unique pipe ID. In this case, the `discoverRestoServices()` method will need to be modified to search peergroup advertisements for all known members of the RestoNet peergroup.

Since the RestoPeer service is now part of the RestoNet peergroup advertisement, HungryPeers will automatically instantiate the service when they join the peergroup. A more sophisticated implementation of the RestoPeer peergroup service would allow each peer to select which service
mode it uses—for example, the RestoPeer service could have a RestoPeer mode and a HungryPeer mode.

Note the important ramifications of this last point. When you run the HungryPeer, the RestoPeerService class must be in your classpath (since the HungryPeer will attempt to instantiate the peergroup service). When the HungryPeer sends out a request, it will get a response from itself (i.e., it will get a response from the HandleFriesRequestThread thread that was started when the HungryPeer initialized the service). This is how peergroup services are designed to work (and clearly, they are more appropriate for some services than others).

6.5 Integration with Other Network Services

As we said at the start of this chapter, JXTA does not require that network services use pipes or other JXTA facilities in order to be invoked; you can create network services that advertise themselves to JXTA peergroups but are invoked via another technique. We'll conclude this chapter with a brief discussion of how this can be accomplished.

We'll center our discussion around an RMI-based network service. The specific details in our discussion will apply to RMI-based services only, but the general technique will be applicable to other types of network services.

The general technique is this: the service must publish a standard module service advertisement within a JXTA peergroup. The service advertisement must contain information so that JXTA peers can contact it; this information is stored in the param tag of the service advertisement. JXTA peers that discover the advertisement extract the special information from the param tag and contact the service using that information.

RMI clients contact RMI servers via a specialized class known as a stub. An RMI service that wants to advertise itself in a JXTA peergroup must encode its stub into the service advertisement. Here's how a service can achieve this:

```java
import java.rmi.*;
import java.rmi.server.*;
import java.io.*;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.exception.PeerGroupException;
import net.jxta.document.Advertisement;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.discovery.DiscoveryService;
import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.protocol.ModuleClassAdvertisement;
import net.jxta.protocol.ModuleSpecAdvertisement;
import net.jxta.platform.ModuleClassID;
import net.jxta.id.IDFactory;
import jxta.security.util.URLBase64;

public class RMIServiceImpl implements RMIService {
    PeerGroup netpg;
    String stub;

    public RMIServiceImpl(  ) throws RemoteException { }
```
public void init( ) throws Exception {
    try {
        // Discover and join (or start) the default peer group
        netpg = PeerGroupFactory.newNetPeerGroup( );
    } catch (PeerGroupException e) {
        // Couldn’t initialize; can’t continue
        System.out.println("Fatal error : creating the NetPeerGroup");
        System.exit(1);
    }
    makeStub( );
    publishAdv( );
}

// The stub is how RMI clients call RMI servers. RMI clients normally
// get the stub from the lookup service via object serialization; in
// this example, they’ll get the stub from the module service
// advertisement. Note that the serialized object is
// encoded as an ASCII
// string so that it can be transmitted through a variety
// of networks.
private void makeStub( ) throws IOException {
    ByteArrayOutputStream baos = new ByteArrayOutputStream( );
    ObjectOutputStream oos = new ObjectOutputStream(baos);
    oos.writeObject(UnicastRemoteObject.exportObject(this));
    oos.close( );
    stub = new String(URLBase64.encode(baos.toByteArray( )));
}

public String sayHello( ) {
    return "Hello";
}

// Publish the advertisement. There’s no point in caching this
// advertisement because the RMI stub is invalid between runs. So we
// always have to publish a new advertisement.
public boolean publishAdv( ) {
    try {
        DiscoveryService disco = netpg.getDiscoveryService( );
        ModuleClassAdvertisement mca = (ModuleClassAdvertisement)
            AdvertisementFactory.newAdvertisement( ModuleClassAdvertisement.getAdvertisementType( )
        );
        mca.setName("JXTAMOD:RMIService:HelloService");
        mca.setDescription("Sample RMI-as-JXTA Service");
    } catch (PeerGroupException e) {
        // Couldn’t initialize; can’t continue
        System.out.println("Fatal error : creating the NetPeerGroup");
        System.exit(1);
    }
    return true;
}
ModuleClassID mcID = IDFactory.newModuleClassID();
mca.setModuleClassID(mcID);
disco.publish(mca, DiscoveryService.ADV);
disco.remotePublish(mca, DiscoveryService.ADV);

ModuleSpecAdvertisement msa = (ModuleSpecAdvertisement)
AdvertisementFactory.newAdvertisement(
ModuleSpecAdvertisement.getAdvertisementType());
msa.setName("JXTASPEC:RMIService:HelloService");
msa.setVersion("Version 1.0");
msa.setCreator("sun.com");
msa.setSpecURI("http://www.jxta.org/tutorial/RMIService.jar");
msa.setModuleSpecID(IDFactory.newModuleSpecID(mcID));
StructuredTextDocument doc = (StructuredTextDocument)
StructuredDocumentFactory.newStructuredDocument(
  new MimeMediaType("text/xml"), "Parm");
Element e = doc.createElement("Stub", stub);
doc.appendChild(e);
msa.setParam(doc);
disco.publish(msa, DiscoveryService.ADV);
disco.remotePublish(msa, DiscoveryService.ADV);
System.out.println("Created the RMI advertisement");
} catch (Exception e) {
  System.out.println(e);
e.printStackTrace(  );
  return false;
}
return true;

public static void main(String[] args) throws Exception {
  RMIServiceImpl rsi = new RMIServiceImpl(  );
  rsi.init(  );
  System.out.println("RMIService waiting for requests...");
  // If we exit, RSI becomes eligible for GC, which means
  // it can no longer handle requests
  synchronized(rsi) {
    rsi.wait(  );
  }
}

For the most part, this example is the same as the RestoNet peer that we presented at the beginning of the chapter. The difference is that after we create the module specification advertisement (and before we publish it), we insert a special document into the param tag of the
advertisement. This XML document is created with a tag name of Parm; its value contains the serialized stub that the client will use to contact the service. Here’s how a client accesses the service:

```java
import java.util.*;
import java.io.*;

import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.exception.PeerGroupException;
import net.jxta.document.Advertisement;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.MimeMediaType;
import net.jxta.document.TextElement;
import net.jxta.discovery.DiscoveryService;
import net.jxta.protocol.ModuleSpecAdvertisement;

import jxta.security.util.URLBase64;

public class RMIClientPeer {
    static Enumeration peers;
    static PeerGroup netpg;

    static int timeout = 10000;
    static int count = 3;

    public static void main(String[] args) throws Exception {
        try {
            netpg = PeerGroupFactory.newNetPeerGroup();
        } catch (PeerGroupException pge) {
            // Couldn’t initialize; can’t continue
            System.out.println("
                Fatal error : creating the NetPeerGroup");
            System.exit(-1);
        }
        if (!discoverRMIPeers()) {
            System.out.println("Can’t find RMI peers");
            System.exit(-1);
        }
    }

    private static boolean discoverRMIPeers() {
        DiscoveryService disco = netpg.getDiscoveryService();
        disco.getRemoteAdvertisements(null,
            "Name",
            "JXTASPEC:RMIService:HelloService", 5, null);
        try {
            Thread.sleep(timeout);
        } catch (InterruptedException ie) {} 

        System.out.println("Looking for RMI Service peers...");
        while (count-- > 0) {
            try {
                disco = netpg.getDiscoveryService();
                disco.getRemoteAdvertisements(null,
                    DiscoveryService.ADV,
                    "Name",
                    "JXTASPEC:RMIService:HelloService", 5, null);
                for (Enumeration e = disco.getRemoteAdvertisements();
                    e.hasMoreElements();
                    System.out.println("Found RMI Service peer: "
                        + e.nextElement().toString());
            }
```
try {
    peers = disco.getLocalAdvertisements(
        DiscoveryService.ADV,
        "Name",
        "JXTASPEC:RMIService:HelloService");
    if (peers != null && peers.hasMoreElements( ))
        break;
    disco.getRemoteAdvertisements(null,
        DiscoveryService.ADV,
        "Name",
        "JXTASPEC:RMIService:HelloService", 5, null);
    try {
        Thread.sleep(timeout);
    } catch (InterruptedException ie) {} 
} catch (Exception e) {
    // Try again
}

if (peers == null || !peers.hasMoreElements( ))
    return false;
return true;
}

private static void callPeers( ) {
    while (peers.hasMoreElements( )) {
        try {
            Object o = peers.nextElement( );
            ModuleSpecAdvertisement msa =
                (ModuleSpecAdvertisement) o;
            StructuredTextDocument doc =
                (StructuredTextDocument) msa.getParam( );
            if (doc == null) {
                // No params
                System.out.println("RMIService adv has no params; ignoring");
                continue;
            }
            Enumeration elements = doc.getChildren( );
            String stub = null;
            while (elements.hasMoreElements( )) {
                TextElement te =
                    (TextElement) elements.nextElement();
                String elementName = te.getName( );
                if (elementName.equals("Stub")) {
                    stub = te.getTextValue( );
                    break;
                }
            }
            if (stub == null) {
                System.out.println("No Stub in RMIService adv");
            } else {
                String stubName = stub.toString();
                if (stubName.length() > 0) {
                    System.out.println(stubName);
                }
            }
        } catch (Exception e) {
            System.out.println("Error while processing RMIService adv");
        }
    }
}
"Didn't find a stub parameter; ignoring");
  continue;
}
byte[] enc = stub.getBytes( );
ByteArrayInputStream bais =
new ByteArrayInputStream(URLBase64.decode(enc, 0, enc.length));
ObjectInputStream ois = new
ObjectInputStream(bais);
RMIService rs = (RMIService) ois.readObject( );
  System.out.println("Remote service says "+ rs.sayHello( ));
} catch (Exception e) {
  System.out.println("Couldn't talk to peer -- "+ e);
  e.printStackTrace( );
}
}

The interesting part of this example is the reverse of what we saw in the service: here the peer must extract the param tag from the service message, deserialize the remote stub contained in that string, and then use that stub to call the service. Even though the peer located the service using standard JXTA protocols, the communication in the sayHello( ) method uses only RMI calls; once the peers have located each other, they communicate using their native network service protocols.

6.5.1 Running This Example

This example is run just like the other examples in this book. Although it involves the use of RMI, you do not need to run the rmiregistry (and you’ll notice that the RMIService class does not register with the rmiregistry): the stubs are found through JXTA instead.
In order to run the example, the client must have a copy of the RMIServiceImpl_Stub class in its classpath. This is unusual for an RMI client; normally, it would download this class automatically as part of its interaction with the server. However, we obtained the stub through the JXTA advertisement in this example. It is possible to send the class bytecodes of the RMIServiceImpl_Stub class in the JXTA advertisement in the same way we sent the serialized stub; the client could then create a classloader and load the new class. This is a straightforward extension to the example.

6.6 Summary

In this chapter, we examined how JXTA programs may become network services: they may become JXTA peer services, they may become JXTA peergroup services, and arbitrary network services can advertise themselves via JXTA to other JXTA peers.
Network services have many advantages over standard applications: they allow developers to create a set of cooperating, highly available, and robust services. This type of service model, one that can be robust in the face of network failure and scalable as more peers join the network, is one of the key reasons why JXTA is such a powerful technology.
Chapter 7. Security

In this chapter, we discuss the security infrastructure provided by the JXTA platform. From the beginning, JXTA was designed with a security infrastructure in mind. This is a big advantage: history has shown that when security is not included in initial architecture, it is far easier for malicious intruders to exploit any security mechanisms that are later added to the system. It is also always more difficult to add security infrastructure after a system has been designed.

Security in a P2P network presents some interesting challenges due to the distributed computing environment, the vulnerability of links (due to multi-hop routing), the dynamic nature of peer interactions, and the lack of a centralized authority. JXTA security requirements are very similar to traditional systems, but the decentralized nature of JXTA makes it more difficult to implement confidentiality, integrity, and nonrepudiation.

Typically, security models define security in terms of users, objects, and actions. The action defines the operation allowed on an object by a user. The JXTA platform does not provide for the concept of users. This keeps the core minimal; user definition can always be done at a service layer. The JXTA protocols do not need to know who a user is or how a user is authenticated. Instead, the protocol messages provide placeholder fields to store security credentials.

Given the organization of the JXTA platform, one can envision a security architecture in which peer and peergroup IDs are treated as low-level subjects (just like user IDs or group IDs), advertisements are treated as objects (just like files); and actions are operations on peers, peergroups, and advertisements (e.g., the capability of a peer to publish or access an advertisement).

7.1 JXTA Security Framework

The security requirements for JXTA are further affected by some unique characteristics:

- **JXTA focuses on mechanisms and not policy.** For example, UUIDs are used throughout JXTA, but by themselves they have no external meaning. Without additional naming and binding services, UUIDs are just numbers that do not correspond to anything like a user or a principal. Therefore, unlike some other computer systems, JTXA does not define a high-level security model, such as information flow, Bell-LaPadula, or Chinese Wall. When UUIDs are bound to external names or entities to form security principals, binding authenticity can be ensured by placing in the data field security attributes, such as digital signatures, that testify to the trustworthiness of the binding. Once this binding is established, we can authenticate the principal, access control based on the principal and the prevailing security policy, and perform other functions, such as resource usage accounting.

- **JXTA is neutral to cryptographic schemes or security algorithms.** As such, JXTA does not mandate any specific security solution. In such cases, JXTA provides a framework that allows different security solutions to be plugged in. For example, every message has a designated credential field that can be used to store security-related information. However, exactly how such information is interpreted is beyond the scope of the specification and is left to services and applications.

- **JXTA can sometimes satisfy security requirements at different levels of the system.** To allow for maximum flexibility and to avoid redundancy, JXTA technology typically does not force a particular implementation on developers. Instead, we expect that a number of enhanced platforms will emerge to provide the appropriate security solutions to their targeted deployment environment.

To illustrate the last point, let’s examine two requirements: communications security and anonymity.

7.1.1 Communications Security
We have seen that peers communicate through pipes. For the sake of discussion, suppose that both confidentiality and integrity in the communications channel are desired. This means that the parties on each end of the pipe must identify themselves; this provides the desired integrity of the communication. Data that is sent between the parties is always encrypted, which provides the desired confidentiality.

One common approach to this task is to use a virtual private network (VPN). Another solution is to create a secure version of the pipe, similar to a protected tunnel, so that any message transmitted over this pipe is automatically secured. A third solution is to use regular communication mechanisms but to let the developer protect specific data payloads with encryption techniques and digital signatures. JXTA technology can accommodate any of these solutions. The VPN runs outside of (and transparent to) the JXTA platform, secure JXTA unicast pipes correspond to the second solution, and the APIs we’ll examine later in this chapter can be used for the third solution.

### 7.1.2 Anonymity

Anonymity does not mean the absence of identity. Sometimes a certain degree of identification is unavoidable. For example, a cell phone number or a SIM card identification number cannot be kept anonymous because it is needed by the telephone company to authorize and set up calls. Similarly, the IP number of a computer cannot be hidden from its nearest gateway or router if the computer wants to send and receive network traffic.

Anonymity can be built on top of identity, but not vice versa. You can ensure anonymity in many ways. In the previous examples, it is difficult to link a pre-paid SIM card, sold over the retail counter for cash, to an actual cell phone user. Likewise, a cooperative gateway or router can help hide the computer’s true IP address from the outside world by using message relays or NAT.

Along these same lines, a JXTA service can bind a peer to a human user, ensuring the user’s anonymity through the naming service, the authentication service, a proxy service, or any combination thereof. JXTA is independent of the solution chosen by a particular application.

### 7.2 JXTA Cryptography

Because each peer may act as a relay to forward messages between two peers, a peer must protect itself against a malicious relay peer that disseminates fallacious routing information or modifies data that passes through it. End-to-end communication channels need to be secured within the JXTA multi-hop network. This means that JXTA peers need to use appropriate cryptographic keys. A typical solution is to embed the cryptographic information in a smart card (SIM card) for handheld devices that may risk being lost or stolen.

JXTA provides a basic set of security classes based on the Java Card security 2.1 platform APIs. The Java Card API provides a minimal infrastructure appropriate for small, mobile, wireless devices, such as PDAs and cell phones. The JXTA security classes provide the basic foundation for defining RSA and secret keys; performing RC4 encryption; creating SHA-1 and MD5 message digests; and creating secure hashes and digital signatures based on these digests and an SHA-1-based, pseudo-random number generator. In the next few sections, we’ll look at JXTA’s Java API bindings for these operations, including an example that shows how this API can be used to sign information contained within a JXTA document. We’ll conclude with a discussion of the security considerations of JXTA’s membership requirements.

### APIs and Implementations

The language bindings for JXTA are somewhat incomplete as of this writing. There is a standard set of API classes in the `jxta.security` package; like most API packages, the definitions in this package tend to be abstract classes and interfaces. Sun's JXTA includes a set of implementation classes in the `jxta.security.impl` package; these classes implement the standard APIs. Typically, you are not expected to directly use classes in the implementation package. However, in the case of the JXTA security API, this cannot be avoided. Therefore,
our examples in this chapter, we'll primarily use the implementation classes. Along the way, we'll discuss the logical operations that are taking place so that when the security API is completed, you'll be able to use that API as intended. Consult the jxta.org site for further developments.

As we proceed through this chapter, you'll notice that the JXTA cryptographic APIs are very different from the security and cryptographic APIs that are used by the Java 2 Standard Edition (J2SE) platform and its various extensions. JXTA follows the Java Card platform APIs because JXTA applications must be able to run on a variety of platforms, including those based on the Java 2 Micro Edition platform (J2ME). The J2SE APIs require more support from their platform; this is a great benefit for desktop and server applications, but it is impractical for small devices. The API discussion in this chapter assumes some familiarity with cryptographic principles. For a good discussion of how cryptographic algorithms work, see Jonathan Knudsen's Java Cryptography (O'Reilly) or Bruce Schneier's Applied Cryptography (John Wiley & Sons).

7.3 JXTA Keys

The first step in utilizing JXTA's security infrastructure is to obtain the necessary set of keys. The security infrastructure supports and uses two kinds of keys:

**Secret keys**
These are symmetric keys that encrypt and decrypt data. They are symmetric because the same key converts plain-text to ciphertext (encrypted text) and ciphertext to plain-text. The standard JXTA bindings support secret keys for RC4 ciphers.

**Public and private key pairs**
These are asymmetric keys used for a variety of operations. Data that is encrypted (or signed) with a public key can be decrypted only with a corresponding private key, and vice versa. JXTA supports algorithms based on RSA public/private key pairs.

The big difference between these types of keys is how they are shared. Public and private key pairs have a big advantage here: you keep your private key private, but you can share your public key with the whole world. People who want to send you data can use your public key to encrypt the data; they know that only you can decrypt the data because only you have the necessary private key. Conversely, if you send data encrypted with your private key to someone else, she can verify that the data was sent by you: your public key can decrypt the data, which proves that it was initially encrypted with your private key.

Shared keys, on the other hand, must be shared between both participants in a data exchange; however, each must have the same shared key at their disposal. This can be very difficult to arrange. It turns out that cryptographic algorithms using shared keys perform much faster than those using public/private key pairs. Therefore, an effective technique is to use RSA encryption to send a secret key to a peer and to use that secret key for all further cryptography with that peer. It's also possible to use Diffie-Hellman key exchange to obtain a secret key to use.

7.3.1 Key Management

JXTA provides a way to create different kinds of keys (the details of which we'll see later in this section). However, JXTA uses advertisements as a mechanism to store or transmit these keys. In particular, public keys are usually stored in certificates (such as an X.509 certificate), which provide a level of assurance as to the owner of the public key. The standard JXTA bindings do not use certificates. Although certificates are encoded within messages that are passed between JXTA peers over secure pipes, there is no public API that retrieves or manipulates those certificates. Public and private keys are relatively long-lived keys and therefore need to be placed in permanent storage. Similarly, public keys must be transmitted to other parties; if you're going to use an RSA digital signature to sign data that you send to another person, he must have your public key to verify your signature.

You can accomplish any of these tasks through a variety of techniques. JXTA is agnostic in this arena; the security API defines only the lowest-level objects necessary to implement the necessary key management systems. The needs of a key management system will vary depending on the
capabilities of a peer; we expect a variety of systems to develop over time. These systems can then be deployed when appropriate.

### 7.3.2 Key Classes

All keys used by the JXTA Java bindings implement the `Key` interface (`jxta.security.cipher.Key`). This interface defines certain basic methods that are implemented by all keys (including the `getType()` method, which is used to determine the type of key). Public and private keys, respectively, implement the `PublicKey` and `PrivateKey` interfaces (in the `jxta.security.publickey` package). Depending on the type of key, they may implement other interfaces—for example, an RSA public key implements the `RSAPublicKey` interface (`jxta.security.publickey.RSAPublicKey`).

### 7.3.3 The KeyBuilder Class

The `KeyBuilder` class (`jxta.security.impl.cipher.KeyBuilder`) creates `Key` objects. This class can create the following types of keys (defined within the `KeyBuilder` class):

```java
// Available key types
public static final byte TYPE_RSA_PUBLIC = 1;
public static final byte TYPE_RSA_PRIVATE = 2;
public static final byte TYPE_RSA = (TYPE_RSA_PUBLIC+TYPE_RSA_PRIVATE);
public static final byte TYPE_DES = 4;
public static final byte TYPE_RC4 = 8;
```

When you create a key, you must specify the length of the key. The key length determines the cryptographic strength of the underlying algorithm: the bigger the length, the harder it is to use a brute-force attempt to break the cryptographic algorithm. The `KeyBuilder` class supports the following key sizes:

```java
// Key sizes RC4 and RSA are for export, as defined in rfc2246
public static final short LENGTH_RC4 = 128;
public static final short LENGTH_RSA_MIN = 384;
public static final short LENGTH_RSA_512 = 512;
```

The keys produced from the `KeyBuilder` class are uninitialized; they must be initialized before they can be used. The process will vary depending on the algorithm supported by the keys.

### 7.3.4 Creating an RSA Key Pair

An RSA public and private key pair is created this way:

```java
RSAKey rsaKey = (RSAKey) KeyBuilder.buildKey(KeyBuilder.TYPE_RSA,
                                           KeyBuilder.LENGTH_RSA_512, false);
```

There are two ways to initialize an `RSAKey` object, depending on whether you want to generate a new key or whether you're creating the object based on a key you've previously generated and stored.

When you create a new key, call the following methods to generate the public and private keys:

```java
RSAKey rsaKey = (RSAKey) KeyBuilder.buildKey(KeyBuilder.TYPE_RSA,
                                           KeyBuilder.LENGTH_RSA_512, false);
JxtaCrypto jc = new JxtaCryptoSuite(..., rsaKey, ...); // See next section
PublicKeyAlgorithm pka = jc.getJxtaPublicKeyAlgorithm( );
pka.setPublicKey( );
pka.setPrivateKey( );
```
As we've mentioned, RSA keys are long-lived entities. Typically, you save your public and private key in some persistent storage and must send your public key to anyone with whom you've exchanged RSA-encoded information. The keys themselves are extracted from the RSAKey object as follows:

```java
RSAPublickeyData publicKeyData = (RSAPublickeyData) pka.getPublickey(  );
RSAPrivatekeyData privateKeyData = (RSAPrivatekeyData) pka.getPrivatekey(  );
```

To create a key based on saved data, you must read the data into the appropriate variable type (an RSAPublickeyData or RSAPrivatekeyData object). For a public key, you can recreate the RSAKey object like this:

```java
RSAKey rsaKey = (RSAKey) KeyBuilder.buildKey(KeyBuilder.TYPE_RSA,
                                               KeyBuilder.LENGTH_RSA_512, false);
JxtaCrypto jc = new JxtaCryptoSuite(..., rsaKey, ... ); // See next section
PublicKeyAlgorithm pka = jc.getJxtaPublicKeyAlgorithm(  );
pka.setPublicKey(publicKeyData);
```

For a private key, you recreate the RSAKey object like this:

```java
RSAKey rsaKey = (RSAKey) KeyBuilder.buildKey(KeyBuilder.TYPE_RSA,
                                               KeyBuilder.LENGTH_RSA_512, false);
JxtaCrypto jc = new JxtaCryptoSuite(..., rsaKey, ... ); // See next section
PublicKeyAlgorithm pka = jc.getJxtaPublicKeyAlgorithm(  );
pka.setPublicKey(publicKeyData);
pka.setPrivateKey(privateKeyData);
```

Note that you must set the public key before setting the private key. We'll use these code fragments later when we create and verify digital signatures using RSA keys.

### 7.3.5 Creating an RC4 Key

An RC4 key may be created as follows:

```java
SecretKey secretKey = (SecretKey) KeyBuilder.buildKey(
    KeyBuilder.TYPE_RC4,
    KeyBuilder.LENGTH_RC4, false);
```

To initialize a secret key, you must obtain a set of random bytes to use as input to the `setKey()` method of the SecretKey class:

```java
JRandom random = new JRandom(  );
byte[] keydata = new byte[KeyBuilder.LENGTH_RC4 >>> 3];
random.nextBytes(keydata);
secretKey.setKey(keydata, 0);
```

In future examples, we'll see how these keys are actually used.

### 7.4 The JxtaCrypto Interface

Classes that implement the JxtaCrypto interface provide one of the main interfaces to the JXTA security API. This interface contains a number of methods, each of which is used to obtain a class that performs a particular operation. If you're familiar with the security implementation of the Java 2 Standard Edition platform, you can think of the JxtaCrypto interface as a simple provider (except that there is no provider infrastructure; you obtain a JxtaCrypto object directly).

The JxtaCryptoSuite class implements the JxtaCrypto interface.
interface in the bindings that come with JXTA. The idea here is that different JXTA implementations will supply a JXTA crypto class that supports a profile of operations that are suitable for the platform on which the implementation runs. Therefore, the crypto suite is instantiated with information that defines the security profile that the application requires.

The `JxtaCrypto` interface defines the following profiles:

- `static final byte PROFILE_RSA_RC4_SHA1;`
- `static final byte PROFILE_RSA_RC4_MD5;`
- `static final byte PROFILE_RSA_SHA1;`
- `static final byte PROFILE_RSA_MD5;`
- `static final byte PROFILE_RSA_RC4_SHA1_MD5;`
- `static final byte PROFILE_RC4_SHA1;`
- `static final byte PROFILE_RC4_MD5;`

Each constant in the name of a profile determines a particular algorithm that the profile supports: RSA algorithms, RC4 encryption, SHA-1 message digests, and MD5 message digests. Therefore, a crypto suite with a profile that supports RSA algorithms and SHA message digests is constructed like this:

```java
JxtaCryptoSuite jcs = new JxtaCryptoSuite(JxtaCrypto.PROFILE_RSA_SHA1, rsaKey, Signature.ALG_RSA_SHA_PKCS1, (byte) 0);
```

In addition to specifying the profile and RSA key, the constructor requires the type of signature algorithm and type of MAC algorithm you expect the suite to support (note that the profile must also support the algorithms that you specify should be used for the signature and MAC operations). Once you've constructed a crypto suite, you use the suite to obtain objects that perform various operations, as follows:

```
Cipher c = jcs.getJxtaCipher(  );
```

When we introduce cryptographic operations, we'll show how they are obtained from the appropriate crypto suite.

### 7.5 Ciphers

Classes that implement the `Cipher` interface (`jxta.security.cipher.Cipher`) are used to encrypt or decrypt data. Cipher objects are obtained via the `getJxtaCipher()` method of the `JxtaCrypto` interface.

Before it can be used, a cipher must be initialized with an appropriate key and a mode (either `Cipher.MODE_DECRYPT` or `Cipher.MODE_ENCRYPT`). Bytes to be encrypted or decrypted are then fed to the cipher's `update()` method; the last block of data to be fed to the cipher is passed to the cipher's `doFinal()` method.

#### 7.5.1 An RC4 Cipher Example

Here's an example that encrypts and decrypts a simple string using the RC4 cipher:

```java
import jxta.security.cipher.Cipher;
import jxta.security.crypto.JxtaCrypto;
import jxta.security.impl.cipher.RC4Cipher;
import jxta.security.impl.cipher.KeyBuilder;
import jxta.security.impl.cipher.SecretKey;
import jxta.security.impl.random.JRandom;
import jxta.security.impl.crypto.JxtaCryptoSuite;

public class TestRC4Cipher {
    public static void main(String[] args) throws Exception {
        // Code goes here
    }
}
// Step 1: Generate the JxtaCryptoSuite that does only
// RC4 encryption (so all other arguments are not
used)
JxtaCrypto jc = new JxtaCryptoSuite(JxtaCrypto.MEMBER_RC4,
null, (byte) 0, (byte) 0);

// Step 2: Generate the necessary RC4 key
SecretKey secretKey = (SecretKey) KeyBuilder.buildKey(
KeyBuilder.TYPE_RC4,
KeyBuilder.LENGTH_RC4, false);
JRandom random = new JRandom( );
byte[] keydata = new byte[KeyBuilder.LENGTH_RC4 >>>
3];
random.nextBytes(keydata);
secretKey.setKey(keydata, 0);

// Step 3: Use the RC4 key to initialize the cipher
Cipher c = jc.getJxtaCipher( );
c.init(secretKey, Cipher.MODE_ENCRYPT);

// Step 4: Encrypt the data. Since our data is short,
we
// use only the doFinal( ) method.
byte[] input = "Hello, JXTA".getBytes( );
byte[] ciphertext = new byte[input.length];
c.doFinal(input, 0, input.length, ciphertext, 0);
System.out.println(
    "Got encrypted data " + new String(ciphertext));

// Now we repeat from step 3 to decrypt the string.
Note that
// we must use the same key to initialize the cipher.
c.init(secretKey, Cipher.MODE_DECRYPT);
byte[] plaintext = new byte[ciphertext.length];
c.doFinal(ciphertext, 0, ciphertext.length, plaintext,
0);
System.out.println(
    "Got unencrypted data " + new String(plaintext));
}

7.6 The Signature Class

The Signature class (jxta.security.signature.Signature) creates and verifies
digital signatures. Digital signatures are used to validate that a particular set of data came from a
particular source. In our JXTA restaurant auction, for example, we might require each restaurant
to sign its bid so that the HungryPeer knows that the bid came from the restaurant listed in the bid.
Otherwise, a competitor to Chez JXTA may send out a bid saying that Chez JXTA’s price for
small fries is $100; without a digital signature to verify the author of the bid document, the
HungryPeer must proceed based only on his trust of the (inherently non-trustworthy) network.
Digital signatures require RSA public and private keys: a digital signature is created with a private
key and verified with a public key. Therefore, in our restaurant example, the HungryPeer must
have Chez JXTA’s public key in order to validate the accompanying signature.
Signature objects are returned via the `getJxtaSignature()` method of the `JxtaCrypto` interface; the type of object that is returned will depend on the profile that was used to instantiate the crypto suite. The valid signature types are defined in the signature class:

```java
static final byte ALG_RSA_SHA_PKCS1;
static final byte ALG_RSA_MD5_PKCS1;
```

Hence, the JXTA platform can work with RSA signatures that use either the MD5 or SHA1 hash algorithm and PKCS1 padding.

Once you have a signature object, you must initialize it either for signing (`Signature.MODE_SIGN`) or for verifying (`Signature.MODE_VERIFY`) a signature. Data that will be signed (or verified) is then fed in blocks to the `update()` method of the signature object. The final block of data is fed to the `sign()` or `verify()` method, which completes the operation.

Here's a short example that illustrates how this works. In this example, we show the necessary steps for adding a digital signature to the bid document that our RestoPeer sends, as well as the steps that the HungryPeer must take in order to verify the digital signature:

```java
public class TestSignature {
    static RSAPublickeyData publicKeyData;
    static RSAPrivatekeyData privateKeyData;

    public static int getDataToSign(StructuredDocument doc, byte[] buf) throws Exception {
        int length = 0;
        Enumeration enum = doc.getChildren("Price");
        while (enum.hasMoreElements()) {
            Element element = (Element) enum.nextElement();
            String value = (String) element.getValue();
            byte[] temp = value.getBytes();
            System.arraycopy(temp, 0, buf, length, temp.length);
            length += temp.length;
        }
        enum = doc.getChildren("Brand");
```
while (enum.hasMoreElements()) {
    Element element = (Element) enum.nextElement();
    String value = (String) element.getValue();
    byte[] temp = value.getBytes();
    System.arraycopy(temp, 0, buf, length, temp.length);
    length += temp.length;
}
return length;
}

public static boolean validateMessage(StructuredDocument doc)
    throws Exception {
    // Step 1: Generate and initialize the necessary RSA key
    RSAKey rsaKey = (RSAKey) KeyBuilder.buildKey(KeyBuilder.TYPE_RSA,
        KeyBuilder.LENGTH_RSA_512, false);

    // Step 2: Create a profile that can do signing
    JxtaCrypto jc = new JxtaCryptoSuite(JxtaCrypto.PROFILE_RSA_SHA1,
        rsaKey, Signature.ALG_RSA_SHA_PKCS1, (byte) 0);

    // Step 3: Initialize the key based on the saved data
    // (e.g., you’d normally read this data from persistent store).
    // Since we’re validating the signature, use the public key.
    PublicKeyAlgorithm pka = jc.getJxtaPublicKeyAlgorithm();
    pka.setPublicKey(publicKeyData);

    // Step 4: Get the data from the document. In this case, we also
    // need the signature data.
    byte[] data = new byte[1024];
    int length = getDataToSign(doc, data);

    // Get the signature; decode it from an ASCII string into the
    // necessary byte array
    Enumeration enum = doc.getChildren("Signature");
    byte[] signature = null;
    while (enum.hasMoreElements()) {
        Element element = (Element) enum.nextElement();
        String value = (String) element.getValue();
        byte[] enc = value.getBytes();
        signature = URLBase64.decode(enc, 0, enc.length);
public static void signMessage(StructuredDocument doc) throws Exception {

    // Step 1: Generate and initialize the necessary RSA key
    RSAKey rsaKey = (RSAKey) KeyBuilder.buildKey(KeyBuilder.TYPE_RSA,
        KeyBuilder.LENGTH_RSA_512,
        false);

    // Step 2: Create a profile that can do signing
    JxtaCrypto jc = new JxtaCryptoSuite(JxtaCrypto.PROFILE_RSA_SHA1,
        rsaKey,
        Signature.ALG_RSA_SHA_PKCS1,
        (byte) 0);

    // Step 3: Initialize the key based on the saved data (e.g.,
    // you’d normally read this data from persistent store)
    PublicKeyAlgorithm pka = jc.getJxtaPublicKeyAlgorithm();
    pka.setPublicKey( );
    pka.setPrivateKey(privateKeyData);

    // Step 4: Get the signature object, initialize it, and sign the
    // data
    Signature s = jc.getJxtaSignature( );
    s.init(Signature.MODE_SIGN);
    byte[] data = new byte[1024];
    int length = getDataToSign(doc, data);
    byte[] signature = s.sign(data, 0, length);

    // Step 5: Add the signature to the document. We first convert it
    // to ASCII so that it can be transmitted over any network.
    Element el = doc.createElement("Signature", "Signature",
        signature, 0, signature.length);

    return verified;
}
new String(URLBase64.encode(signature));
    doc.appendChild(el);
}

public static void main(String[] args) throws Exception {
    // Step 1: Generate and initialize the necessary RSA key
    RSAKey rsaKey = (RSAKey)
        KeyBuilder.buildKey(KeyBuilder.TYPE_RSA,
            KeyBuilder.LENGTH_RSA_512,
            false);

    // Step 2: Create a profile that can do signing
    JxtaCrypto jc = new JxtaCryptoSuite(JxtaCrypto.PROFILE_RSA_SHA1,
        rsaKey,
        Signature.ALG_RSA_SHA_PKCS1,
        (byte) 0);

    // Step 3: Complete the initialization of the keys
    PublicKeyAlgorithm pka = jc.getJxtaPublicKeyAlgorithm();
    pka.setPublicKey(  );
    pka.setPrivateKey(  );

    // Step 4: Save the public/private key data. Normally, these would
    // go into persistent storage, and the HungyPeer would
    // be sent the
    // public key.
    publicKeyData = (RSAPublickeyData) pka.getPublickey(  );
    privateKeyData = (RSAPrivatekeyData) pka.getPrivatekey(  );

    // Step 5: Create the document, just as we did in the RestoPeer
    StructuredDocument bid =
        StructuredDocumentFactory.newStructuredDocument(
            new MimeMediaType("text", "xml"),
            "RestoNet:Bid");
    Element el = bid.createElement("Brand", "Chez JXTA");
    bid.appendChild(el);
    el = bid.createElement("Price", "$1.00");
    bid.appendChild(el);

    // Step 6: Sign the document
    signMessage(bid);

    // Step 7: Validate the document. We’d normally transmit the
    // document, and the recipient would validate its signature. We’ll
    // just do this instead.
    boolean valid = validateMessage(bid);}
System.out.println("Signature was " + ((valid) ? "ok" : "invalid"));
}

### 7.7 The Hash Class

The **Hash class** (jxta.security.hash.Hash) calculates message digests or hashes of data. A message digest is a short array of bytes that is calculated based on an arbitrary set of data; you can take millions of bytes of data and calculate a 64-byte hash of the data. Hashes are not guaranteed to be unique, but the odds that two sets of data produce the same hash are quite remote (1 in \(2^{64}\)).

By themselves, hashes are not secure. You can transmit a hash value along with some data, and the recipient of the data can recreate the hash. If the newly calculated hash does not match the original hash, then the receiver knows that the data was modified in transit. However, if the hash is sent with the data, then nothing prevents the hash from being modified along with the data, leaving the receiver unaware of the manipulation.

There are two ways to solve this problem. The first is to use a digital signature. A digital signature is essentially (though not exactly the same thing as) an encryption of the hash value. Because anyone who modifies the data used to create the hash lacks the necessary key to produce the digital signature, the signed hash is secure; if the digital signature verification succeeds, you know that the data was not modified in transit.

A second technique is to use a message authentication code (MAC), which we'll look at in the next section.

The JXTA API supports two hash algorithms: MD5 and SHA1. Hash objects are obtained from the `getJxtaHash()` method of the `JxtaCrypto` interface; the algorithm that is used depends on the value used to construct the crypto suite. This value must be one of the following constants from the **Hash** class:

- `public static final byte ALG_SHA1;`  // SHA1 digest
- `public static final byte ALG_MD5;` // MD5 digest

The `Hash` class works the same as the `Signature` class does (except that there are no keys involved): data is fed to the `Hash` object via the `update()` method. The final set of data is fed to the `Hash` object via the `doFinal()` method.

### 7.7.1 The MAC Class

The **MAC class** (jxta.security.mac.MAC) creates a secure hash by encrypting the underlying hash value (using RC4 encryption). Therefore, a MAC is very similar to a digital signature; the primary difference is in the keys used. The RSA digital signature uses an RSA private key to create a signature and an RSA public key to verify it; this allows the public key to be shared freely. The MAC uses a secret key both to create and to verify the MAC; this key can be shared only between the parties that are supposed to have access to the signature.

Procedurally, however, using a MAC object is identical to using a signature object with the following exceptions:

- The MAC object itself is returned from the `getJxtaMAC()` method of the `JxtaCrypto` object.
- The MAC object is initialized with a secret key and the bytes required to initialize the secret key:
  - `MAC mac = jc.getJxtaMAC();`
  - `SecretKey secretKey = (SecretKey) KeyBuilder.buildKey(`
    - `KeyBuilder.TYPE_RC4,`
      - `KeyBuilder.LENGTH_RC4,`
        - `false);`
7.8 Secure JXTA Pipes

In our previous examples, we used an unsecure unicast pipe to transmit messages between peers. Such pipes have three potential problems:

- Data on the pipe is sent in the clear. People who can tap into the network can read data that is transmitted on the pipe.
- Data on the pipe may flow through a computer that modifies the data in transit.
- There is no assurance that a pipe is connected to who you think: the peer may identify itself as O'Reilly & Associates but may in fact be someone else.

Secure pipes overcome these limitations by using digitally signed data to identify the peers connected to them. Also, they encrypt data that flows through them. The signature verifies the identity of the connected peer; the encryption prevents data from being read or modified in transit. JXTA Java bindings implement secure pipes using Transport Layer Security (TLS). TLS is compatible with the Secure Socket Layer (SSL) protocol. The SSL protocol was developed by Netscape Communications and went through several versions. Version 3.0 of SSL was contributed to the Internet Engineering Task Force (IETF), which made slight modifications to the protocol and issued the TLS 1.0 specification. TLS implementations are backwards-compatible with SSL 3.0.

When you want to create a secure pipe, set its type like this:
```
myAdv.setType(PipeService.UnicastSecureType);
```

TLS pipes have the following advantages:

- Data sent over the pipe is encrypted; it cannot be read or modified in transit. TLS supports a number of encryption suites, but the JXTA implementation uses `TLS_RSA_WITH_3DES_EDE_CBC_SHA`. This means that JXTA uses an RSA public/private key protocol to initialize the connection, it uses triple-DES in cipher block chaining mode to encrypt data that flows over the pipe, and its digital signatures use the secure hash algorithm. Triple-DES uses a 192-bit key, making it very hard to break.
- During initialization, each peer on the pipe presents its public key certificate to the other peer, which then validates the certificate to confirm its partner’s identity. The certificates that JXTA uses are issued by the JXTA platform itself: JXTA creates a self-signed root certificate on behalf of the platform, and then issues the peer a certificate based on that root certificate.

There’s a certain degree of tension in the certificate-handling here: well-known certificate authorities are generally centralized, but JXTA involves decentralized resources. Using the platform as the built-in certificate authority is a reasonable entry-level compromise: it provides the necessary infrastructure to support TLS. Because the platform is its own certificate authority, the level of assurance you have that a peer is actually who it claims to be is not very high; it is, however, comparable to the level of assurance that you have when presented with the free certificates issued by many other certificate authorities.

Remember, however, that this use of certificates all happens at the application layer, not the protocol layer. The default JXTA bindings can be easily modified to use a commercial certificate authority as the root certificate (perhaps bundling the CA root certificate with the JXTA binaries, as most browsers do) or to adopt any other certificate infrastructure they desire.

7.9 User Credentials
In Chapter 2, we introduced the JTXA configurator and noted that it creates a pse directory that contains user configuration information. Let’s look at that directory in a little more detail.

The pse directory contains four files:

- **pse/client/peer-service.pem**
  This file contains the X.509 certificate that the peer will use when it connects to an existing secure pipe. This certificate is issued by the root certificate.

- **pse/client/peer.phrase**
  This file contains the encrypted password the user entered into the configurator.

- **pse/etc/passwd**
  This file contains the encrypted username and password.

- **pse/root/peer-root.pem**
  This file contains the X.509 certificate that the peer will use when it creates a secure pipe. It is also the root certificate of the client’s peer certificate.

Information in the pse directory is used only by the secure pipe implementation. In fact, only the secure pipe implementation has access to this information; the certificates, username, and password are not available to the rest of the application in any form (unless the application wants to read these files directly and determine how to extract the relevant information from them).

When you follow the examples earlier in this book, you are always prompted for a username and password (or you specify one via command-line properties). The platform will verify the validity of your password; if incorrect, you’ll get an IOException when creating a secure pipe.

The certificate files contain a standard encoding of the X.509 certificate; you can copy the file, edit out the other information, and then feed the encoded certificate to a variety of tools. In particular, such a file can be utilized by the Java 2 Standard Edition security APIs.

### 7.10 JXTA Authentication

In our examples so far, peers have been able to join peergroups without facing any restrictions. The Peer Membership Protocol, however, allows peergroups to restrict their membership by requiring new peers to present a set of credentials. The peergroup then examines these credentials and determines if the peer will be allowed to join the peergroup.

In order to perform this authentication, you must have a class that extends the abstract Membership class (net.jxta.membership.Membership). This class implements the peergroup’s policy with respect to membership: peers that want to join the peergroup are responsible for supplying a set of credentials when they join the group. The membership class examines these credentials and determines if they are valid; if they are, the peer is allowed to join the group.

One implementation of the membership class is the PasswdMembership class (net.jxta.impl.membership.PasswdMembership). This class is initialized with a list of users and their passwords; peers that want to join a peergroup using password membership must supply the credentials with the appropriate password.

Let’s enhance our RestoNet example so that the RestoNet peergroup requires a password in order to join it. This requires changes both to the RestoPeer (which must set up the peergroup with the appropriate membership policy) and to the HungryPeer (which must now supply credentials in order to join the peergroup).

We’ll start with the RestoPeer. In our previous examples, the RestoNet peergroup was created by copying all of the default services from the NetPeerGroup; this was accomplished by using a default peergroup advertisement. In this example, we must supply a new peergroup advertisement that copies most of those services but that includes a new definition for the membership service. Here’s the code to do this:

```java
import java.io.*;
import java.util.*;
import java.net.URL;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.exception.PeerGroupException;
```
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.Advertisement;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredTextDocument;
import net.jxta.discovery.DiscoveryService;
import net.jxta.pipe.PipeService;
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.PipeID;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;
import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.endpoint.Message;
import net.jxta.id.IDFactory;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.membership.MembershipService;
import net.jxta.membership.Authenticator;
import net.jxta.credential.AuthenticationCredential;
import net.jxta.impl.peergroup.StdPeerGroupParamAdv;
import net.jxta.impl.id.UUID.ModuleClassID;
import net.jxta.impl.membership.PasswdMembershipService;

// RestoPeer represents a restaurant that receives auction requests for
// French fries from HungryPeers. RestoPeer offers three sizes of French
// fries (small, medium, large). Each restaurant assigns a different price
// for each size and offers a special. Each restaurant is uniquely
// identified by its brand name.

public class RestoPeer {

    private PeerGroup netpg = null;  // The NetPeerGroup
    private PeerGroup restoNet = null;  // The RestoNet peer group

    private String brand = "Chez JXTA";  // Brand name of this
                                          // restaurant

    private String specials = "large ($3.00)";  // Current special

    // Services within the RestoNet peer group
    private DiscoveryService disco = null;  // Discovery service
    private PipeService pipes = null;  // Pipe service
    private PipeAdvertisement myAdv = null;  // My RestoPeer pipe
                                          // advertisement
private InputPipe pipeIn = null; // Pipe that we listen to
private int timeout = 3000; // for requests
private int rtimeout = 8000; // Discovery wait time-out
private ModuleClassID membershipClassID;
static {try {
    restoPeerGroupID = (PeerGroupID) IDFactory.fromURL(new URL("urn", ",", ",urn:uuid-058C59E291174BEFB804AC9C6A3B5002");
} catch (Exception e) {
    throw new RuntimeException("Can’t load RestoPeer");
}
}

public static void main(String args[]) {
    RestoPeer myapp = new RestoPeer();
    myapp.startJxta();
    System.exit(0);
}

  // Method to start the JXTA platform, join the RestoNet peer group, and
  // advertise the RestoPeer service
  private void startJxta() {
    // ... unchanged from previous examples ...
  }

  // Discover (or create) and join the RestoNet peer group
  private void joinRestoNet() throws Exception {
    int count = 3; // Maximum number of attempts to discover
    System.out.println(
      "Attempting to Discover the RestoNet PeerGroup");
    DiscoveryService hdisco = netpg.getDiscoveryService();
    Enumeration ae = null; // Holds the discovered peers
    // Loop until we discover the RestoNet or until we’ve
    // exhausted the desired number of attempts
    while (count-- > 0) {
      try {
        // search first in the peer local cache to find
        // ... unchanged from previous examples ...
      }
    }
  }
}
// the RestoNet peergroup advertisement
ae = hdisco.getLocalAdvertisements(
    DiscoveryService.GROUP, "Name",
    "RestoNetAuth");

// If we found the RestoNet advertisement, we are done
if ((ae != null) && ae.hasMoreElements( ))
    break;

// If we did not find it, we send a discovery request
hdisco.getRemoteAdvertisements(null, DiscoveryService.GROUP, "Name",
    "RestoNetAuth", 1, null);

    // Sleep to allow time for peers to respond to the discovery request
    try {
        Thread.sleep(timeout);
    } catch (InterruptedException ie) {} catch (IOException e) {
        // Found nothing! Move on
    }

PeerGroupAdvertisement restoNetAdv = null;

    // Check if we found the RestoNet advertisement. If we didn’t, then
    // either we are the first peer to join or no other RestoNet peers
    // are up. In either case, we must create the RestoNet peergroup.

    if (ae == null || !ae.hasMoreElements( )) {
        System.out.println("Could not find the RestoNet peergroup; creating one");
        try {
            // Create a new, all-purpose peergroup.
            Because we have a custom peergroup advertisement, we must create the module
            // implementation advertisement and then the peergroup
            // advertisement (just as we did when we created the
            // peergroup as a service).
            ModuleImplAdvertisement implAdv = createAuthPeerGroupModuleImplAdv(netpg);
            hdisco.publish(implAdv, DiscoveryService.ADV, PeerGroup.DEFAULT_LIFETIME,
                PeerGroup.DEFAULT_EXPIRATION);
hdisco.remotePublish(implAdv,
DiscoveryService.ADV,
  PeerGroup.DEFAULT_EXPIRATION);
restoNetAdv =
createPeerGroupAdvertisement(implAdv,
"RestoNetAuth");

  // The peergroup advertisement membership
  // service must have
  // a parameter that initializes the valid
  // users and
  // passwords. Create a document that holds
  // this information.
  // There can be any number of login:passwd:
  // pairs in the
  // login tag value.
StructuredTextDocument pwds =
(StructuredTextDocument)
StructuredDocumentFactory.newStructuredDocument(
   new MimeMediaType("text/xml"), "Parm");
Element e = pwds.createElement("login",
   "hungrypeer:" +
PasswdMembershipService.makePsswd("password") + ":");
pwds.appendChild(e);
restoNetAdv.putServiceParam(membershipClassID,
pwds);

  // Now publish the modified advertisement as
  usual
hdisco.publish(restoNetAdv,
DiscoveryService.GROUP,
  PeerGroup.DEFAULT_LIFETIME,
  PeerGroup.DEFAULT_EXPIRATION);
hdisco.remotePublish(restoNetAdv,
DiscoveryService.GROUP,
  PeerGroup.DEFAULT_EXPIRATION);
restoNet = netpg.newGroup(restoNetAdv);
} catch (Exception e) {
  System.out.println(
    "Error in creating RestoNet Peergroup " +
e);
  throw e;
}
} else {
  // The RestoNet advertisement was found in the
  // cache;
  // this means we can join the existing RestoNet
  peergroup
  try {
    restoNetAdv = (PeerGroupAdvertisement)
    ae.nextElement(  );
    restoNet = netpg.newGroup(restoNetAdv);
    System.out.println(  )
  }
  catch (Exception e) {
      System.out.println("Error in creating RestoNet Peergroup " +
   e);
      throw e;
  }
"Found the RestoNet Peergroup advertisement";
} catch (Exception e) {
    System.out.println("Error in creating RestoNet PeerGroup from existing adv");
    throw e;
}

// Instead of just joining, we must also authenticate authenticateAndJoin(restoNet);

try {
    // Get the discovery and pipe services for the RestoNet
    disco = restoNet.getDiscoveryService();
    pipes = restoNet.getPipeService();
} catch (Exception e) {
    System.out.println("Error getting services from RestoNet");
    throw e;
}

System.out.println("RestoNet Restaurant (" + brand + ") is on-line");
return;
}

// Method to handle fries auction requests from HungryPeers.
private void handleFriesRequest() {
    // ... unchanged from previous examples ...
}

// Create the resto pipe associated with this RestoPeer private boolean createRestoPipe() {
    // ... unchanged from previous examples ...
}

// Create a ModuleImplAdvertisement that represents a peergroup that is a clone of the NetPeerGroup but has a different membership service
private ModuleImplAdvertisement createAuthPeerGroupModuleImplAdv(PeerGroup parent) {
    ModuleImplAdvertisement allPurposePeerGroupImplAdv = null;
    StdPeerGroupParamAdv PeerGroupParamAdv = null;
    try {
        // Clone the parent (NetPeerGroup) implementation to get
        // a module implementation
allPurposePeerGroupImplAdv =
parent.getAllPurposePeerGroupImplAdvertisement(  );
}
catch (Exception e) {
    System.err.println("Cannot get allPurposePeerGroupImplAdv " + e);
    return null;
}

// Get the parameter field that contains all the
// peer group services
// associated with the peer group
try {
    PeerGroupParamAdv = new StdPeerGroupParamAdv(
        allPurposePeerGroupImplAdv.getParam(  ));
} catch (PeerGroupException e) {
    System.err.println("Cannot get StdPeerGroupParamAdv " + e);
    return null;
}

// Get the hashtable containing the list of all the
// peer group services from the parameter advertisements
Hashtable allPurposePeerGroupServicesHashtable =
    PeerGroupParamAdv.getServices(  );

// Replace the membership service in the hashtable.
This entails
// enumerating the known services to find the
// membership service,
// and then replacing its advertisement.
Enumeration services =
    allPurposePeerGroupServicesHashtable.
    keys(  );
while (services.hasMoreElements(  )) {
    Object key = services.nextElement(  );
    ModuleClassID mcid;
    try {
        mcid = (ModuleClassID) key;
    } catch (ClassCastException cce) {
        System.out.println("Not a service; ignoring");
        continue;
    }
    if (mcid.isOfSameBaseClass(
        PeerGroup.refMembershipSpecID)) {
        // Got the membership service. Save its class ID for
        // later and then make an advertisement for our new
        // membership service.
        membershipClassID = mcid;
ModuleImplAdvertisement mia = ModuleImplAdvertisement.getAdvertisementType()
    AdvertisementFactory.newAdvertisement(
        ModuleImplAdvertisement.getAdvertisementType( ));
    mia.setModuleSpecID(
        PasswdMembershipService.passwordMembershipSpecID);
    mia.setCode(
        "net.jxta.impl.membership.PasswdMembershipService");
    mia.setDescription("Authenticated Membership Service");
    StructuredTextDocument doc = StructuredTextDocument
    StructuredDocumentFactory.newStructuredDocument(
        new MimeMediaType("text/xml"), "Comp");
    Element e = doc.createElement("Efmt", "JDK1.4");
    doc.appendChild(e);
    e = doc.createElement("Bind", "V1.0 Ref Impl");
    doc.appendChild(e);
    mia.setCompat(doc);
    mia.setUri("http://www.jxta.org/download/jxta.jar");
    mia.setProvider("sun.com");
    allPurposePeerGroupServicesHashtable.remove(key);
    allPurposePeerGroupServicesHashtable.put(key, mia);
    break;
}

// Update the PeerGroupModuleImplAdv with our new list of peergroup services
// services
    allPurposePeerGroupImplAdv.setParam((Element)
        PeerGroupParamAdv.getDocument(
            new MimeMediaType("text/xml")));

    // Set the new unique specification ID that identifies this new peergroup implementation
    allPurposePeerGroupImplAdv.setModuleSpecID(
        PasswdMembershipService.passwordMembershipSpecID);

    // We are done creating our new peergroup implementation
    return allPurposePeerGroupImplAdv;

    // This utility method is used to create a peergroup advertisement
// associated with a module peergroup implementation

private PeerGroupAdvertisement createPeerGroupAdvertisement{
    // ... unchanged from previous examples ...
}

// Create an authentication document and use it to join the given
// peergroup
private void authenticateAndJoin(PeerGroup pg) {
    try {
        StructuredDocument creds = null;
        // Get the application document from the peergroup
        AuthenticationCredential authCred =
            new AuthenticationCredential(pg, null, creds);
        MembershipService m = (MembershipService)
            pg.getMembershipService();
        // Get the entity used to check the application
document
        Authenticator auth = m.apply(authCred);
        PasswdMembershipService.PasswdAuthenticator pwAuth =
            (PasswdMembershipService.PasswdAuthenticator) auth;
        // Fill out the application document
        pwAuth.setAuth1Identity("hungrypeer");
        pwAuth.setAuth2_Password("password");
        if (!auth.isReadyForJoin()) {
            throw new IllegalArgumentException("
                Can’t authenticate -- Not ready");
        }
        // Send the application document and join the
        peer group
        m.join(auth);
    } catch (Exception e) {
        System.err.println("Authentication failed: " + e);
        throw new IllegalArgumentException(e.toString);
    }
}

For the most part, this example looks like our example in Chapter 6 that created new peergroup services. The difference is in the createAuthPeerGroupModuleImplAdv( ) method. In our previous example, this method retrieved an all-purpose module advertisement and added a new service to it. What we must do in this example instead is:

1. Get the all-purpose module advertisement.
2. Extract the standard peergroup advertisement from the module advertisement.
3. Extract the services from the peergroup advertisement. The hashtable that is extracted contains an advertisement for each default service of the peergroup; the key in the hashtable is the module class ID for the service. For standard JXTA services, the module class ID is a well-known value.
4. Enumerate through the hashtable of services to find the membership service.
5. Replace the membership services advertisement in the hashtable. This requires a new membership service advertisement, which requires the bulk of the code of the
createAuthPeerGroupModuleImplAdv() method. As with any advertisement, we must populate all the relevant values; note that we set its code value to "net.jxta.impl.membership.PasswdMembershipService."

When the module advertisement is returned to the createRestoNet() method, we must do one more thing: set up the parameters required by the services. In our case, the only service that needs parameters is our new membership service, which requires a parameter document that looks like this:

```xml
<Parm>
  <login>
    hungryuser:********:
  </login>
</Parm>
```

The asterisks in this document will be the encrypted form of the password. The login string itself can have multiple users and passwords, each separated by colons. In a real application, you'd populate this from a password file or naming service (or, better yet, you'd rewrite the PasswdMembership class to find those values), but this gives you the basic idea. Once we add the service parameters to the advertisement, we can publish it and instantiate the peergroup as usual.

When any peer wants to join this peergroup, it must now go through an authentication procedure, which is shown in the authenticateAndJoin() method. There are four steps to authentication:

1. Retrieve the authentication credential document from the peergroup.
2. Get the authenticator that applies to that document.
3. Fill out the authenticator with the appropriate information. (The methods and values that are used here will vary depending on the requirements of the membership implementation. In the case of the PasswdMembershipService class, you must provide a username and password.)
4. Use the authentication document to join the peergroup.

Note that authentication is required by every peer that joins the peergroup. Therefore, we must modify the HungryPeer class to call the same authenticateAndJoin() method before it can participate in the peergroup. In the case of the HungryPeer, you may want to prompt for a username and password. It would be ideal to use the username and password that initialized the JXTA platform, but as we just mentioned, these values are not made public anywhere.

The PasswdMembership class that we used throughout this example is fairly simplistic; it merely checks the presented password against the passwords contained in the parameter string used to initialize the membership service.

Envision a more complex membership service: one that requires passwords to be encrypted, one that requires credentials to be signed, or one that interacts with an LDAP naming service. At a programming level, however, that membership service would exactly follow the steps we see here. The only difference is the methods that will be used to set information within the authenticator object.

### 7.11 Summary

In this chapter, we looked at the basic security-related APIs that come with JXTA. The JXTA protocols are designed with the idea of security: advertisements may be signed, membership within peergroups may be authenticated, and so on. However, JXTA does not impose a particular security model on its peers; we've seen traditional examples in this chapter that use digital signatures and Unix-style passwords. At a protocol level, however, JXTA can support any security model that makes sense for a particular set of peers.
Part II: Quick Reference

Part II contains quick-reference material for the JXTA platform. It includes a reference for the JXTA Shell, a reference for the Java language bindings of the JXTA APIs, and reprints of the JXTA protocol specifications.

Chapter 8
Chapter 9
Chapter 10
Chapter 11
Chapter 12
Chapter 13
Chapter 14
Chapter 15
Chapter 16
Chapter 17
Chapter 18
Chapter 8. How to Use This Quick Reference

The quick-reference section that follows packs a lot of information into a small space. This introduction explains how to get the most out of that information. It describes how the quick reference is organized and how to read the individual quick-reference entries.

8.1 Finding a Quick-Reference Entry

The quick reference is organized into nine chapters. Chapter 9 documents the Java language bindings of the core JXTA platform, Chapter 10 documents the Java language bindings of JXTA's core security features, and Chapter 11 documents the implementation of many of the interfaces from the jxta.security package. Chapter 12 contains a list of commands available within the JXTA Shell. The remaining chapters (Chapter 13-Chapter 18) contain the specifications of the JXTA platform.

Chapter 9 through Chapter 11 begin with a short introduction to the packages document in those chapters. This is followed by a series of sections, each of which covers one package. Each section begins with an overview of the package. Following this overview are quick-reference entries for all of the public classes and interfaces in the package. Entries are organized alphabetically by class and package name, so that related classes are grouped near each other. Thus, in order to look up a quick reference entry for a particular class, you must also know the name of the package that contains that class. Usually, the package name is obvious from the context, and you should have no trouble looking up the quick-reference entry you want. Use the tabs on the outside edge of the book and the dictionary-style headers on the upper outside corner of each page to help you find the package and class you are looking for.

Packages are listed in alphabetical order within each chapter. Note, however, that because of how packages are grouped in each chapter, packages do not appear in strictly alphabetical order. Occasionally, you may need to look up a class and you do not already know its package. In this case, refer to the Class, Method, and Field Index. This index allows you to look up a class by class name and find which package it is a part of.

8.2 Reading a Quick-Reference Entry

Each quick-reference entry contains quite a bit of information. The sections that follow describe the structure of a quick-reference entry, explaining what information is available, where it is found, and what it means. While reading the descriptions, you will find it helpful to flip through the reference section itself to find examples of the features being described.

8.2.1 Class Name, Package Name, Availability, and Flags

Each quick-reference entry begins with a four-part title that specifies the name, package, and availability of the class, and may also specify various additional flags that describe the class. The class name appears in bold at the upper left of the title. The package name appears, in smaller print, in the lower left, below the class name.

The upper-right portion of the title indicates the availability of the class. In the lower-right corner of the title you may find a list of flags that describe the class. The possible flags and their meanings are as follows:

checked

The class is a checked exception, which means that it extends java.lang.Exception, but not java.lang.RuntimeException. In other words, it must be declared in the throws clause of any method that may throw it.

cloneable

The class, or superclass, implements java.lang.Cloneable.

unchecked
The class is an unchecked exception, which means it extends `java.lang.RuntimeException` and therefore does not need to be declared in the `throws` clause of a method that may throw it.

### 8.2.2 Description

The title of each quick-reference entry is followed by a short description of the most important features of the class or interface.

### 8.2.3 Synopsis

The most important part of every quick-reference entry is the class synopsis, which follows the title and description. The synopsis for a class looks a lot like the source code for the class, except that the method bodies are omitted, and some additional annotations are added. If you know Java syntax, you already know how to read the class synopsis.

The first line of the synopsis contains information about the class itself. It begins with a list of class modifiers, such as `public`, `abstract`, and `final`. These modifiers are followed by the `class` or `interface` keyword and then by the name of the class. The class name may be followed by an `extends` clause that specifies the superclass, and an `implements` clause that specifies any interfaces the class implements.

The class definition line is followed by a list of the fields and methods that the class defines. Once again, if you understand basic Java syntax, you should have no trouble making sense of these lines. The listing for each member includes the modifiers, type, and name of the member. For methods, the synopsis also includes the type and name of each method parameter, in addition to an optional `throws` clause that lists the exceptions the method can throw. The member names are in bold, so it is easy to scan the list of members looking for the one you want. The names of method parameters are in italics to indicate that they are not to be used literally. The member listings are printed on alternating gray and white backgrounds to keep them visually separate.

#### 8.2.3.1 Member availability and flags

Each member listing is a single line that defines the API for that member. These listings use Java syntax, so their meanings are immediately clear to any Java programmer. There is some auxiliary information associated with each member synopsis, however, that requires explanation.

The area to the right of the member synopsis is used to display a variety of flags that provide additional information about the member. Some of these flags indicate additional specification details that do not appear in the member API itself. The following flags may be displayed to the right of a member synopsis:

- **synchronized**: An implementation-specific flag that indicates that a method implementation is declared synchronized, meaning that it obtains a lock on the object or class before executing. The `synchronized` keyword is part of the method implementation, not part of the specification, so it appears as a flag, not in the method synopsis itself. This flag is a useful hint that the method is probably implemented in a thread-safe manner. Whether or not a method is thread-safe is part of the method specification, and this information should appear (although it often does not) in the method documentation. There are a number of different ways to make a method thread-safe, however, and declaring the method with the `synchronized` keyword is only one possible implementation. In other words, a method that does not bear the `synchronized` flag might still be thread-safe.

- **Overrides**: Indicates that a method overrides a method in one of its superclasses. The flag is followed by the name of the superclass that the method overrides. This is a specification detail, not an implementation detail. As we'll see in the next section, overriding methods are usually
grouped together in their own section of the class synopsis. The Overrides: flag is used only when an overriding method is not grouped in this way.

**Implements:**
Indicates that a method implements a method in an interface. The flag is followed by the name of the interface that is implemented. This is a specification detail, not an implementation detail. As we'll see in the next section, methods that implement an interface are usually grouped into a special section of the class synopsis. The Implements: flag is used only for methods that are not grouped in this way.

**empty**
Indicates that the implementation of the method has an empty body. This can be a hint to the programmer that the method may need to be overridden in a subclass.

**constant**
An implementation flag indicating that a method has a trivial implementation. Only methods with a void return type can be truly empty. Any method declared to return a value must have at least a return statement. The constant flag indicates that the method implementation is empty except for a return statement that returns a constant value. Such a method might have a body such as return null; or return false; Like the empty flag, this flag indicates that a method may need to be overridden.

**default:**
This flag is used with property accessor methods that read the value of a property (i.e., methods with names that begin with "get" and take no arguments). The flag is followed by the default value of the property. Strictly speaking, default property values are a specification detail. In practice, however, these defaults are not always documented, and care should be taken because the default values may change between implementations. Not all property accessors have a default: flag. A default value is determined by dynamically loading the class in question, instantiating it using a no-argument constructor, and then calling the method to find out what it returns. This technique can be used only on classes that can be dynamically loaded and instantiated and that have no-argument constructors, so default values are shown for these classes only. Furthermore, note that when a class in instantiated using a different constructor, the default values for its properties may be different.

For static final fields, this flag is followed by the constant value of the field. Only constants of primitive and String types and constants with the value null are displayed. Some constant values are specification details, while others are implementation details. The reason that symbolic constants are defined, however, is so you can write code that does not rely directly on the constant value. Use this flag to help you understand the class, but do not rely on the constant values in your own programs.

### 8.2.3.2 Functional grouping of members

Within a class synopsis, the members are not listed in strict alphabetical order. Instead, they are broken down into functional groups and listed alphabetically within each group. Constructors, methods, fields, and inner classes are all listed separately. Instance methods are kept separate from static (class) methods. Constants are separated from non-constant fields. Public members are listed separately from protected members. Grouping members by category breaks a class down into smaller, more comprehensible segments, making the class easier to understand. This grouping also makes it easier for you to find a desired member.

Functional groups are separated from each other in a class synopsis with Java comments, such as // Public Constructors or // Inner Classes. The various functional categories are as follows (in the order in which they appear in a class synopsis):

**Constructors**
Displays the constructors for the class. Public constructors and protected constructors are displayed separately in subgroups. If a class defines no constructor at all, the Java compiler adds a default no-argument constructor, which is displayed here. If a class
defines only private constructors, it cannot be instantiated, so a special, empty grouping
titled "No Constructor" indicates this fact. Constructors are listed first because the first
thing you do with most classes is instantiate them by calling a constructor.

**Constants**
Displays all of the constants (i.e., fields that are declared static and final) defined
by the class. Public and protected constants are displayed in separate subgroups.
Constants are listed here, near the top of the class synopsis, because constant values are
often used throughout the class as legal values for method parameters and return values.

**Inner classes**
Groups together all of the inner classes and interfaces defined by the class or interface.
For each inner class, there is a single-line synopsis. Also each inner class has its own
quick-reference entry that includes a full class synopsis for the inner class. Like constants,
inner classes are listed near the top of the class synopsis because they are often used by a
number of other members of the class.

**Static methods**
Lists the static methods (class methods) of the class, broken down into subgroups for
public static methods and protected static methods.

**Event listener registration methods**
Lists the public instance methods that register and deregister event listener objects with
the class. The names of these methods begin with the words "add" and "remove" and end
in "Listener." These methods are always passed a java.util.EventListener
object. The methods are typically defined in pairs, so the pairs are listed together. The
methods are listed alphabetically by event name, rather than by method name.

**Property accessor methods**
Lists the public instance methods that set or query the value of a property or attribute of
the class. The names of these methods begin with the words "set," "get," and "is," and
their signatures follow the patterns set out in the JavaBeans specification. Although the
naming conventions and method signature patterns are defined for JavaBeans, classes and
interfaces throughout the Java platform define property accessor methods that follow
these conventions and patterns. Looking at a class in terms of the properties it defines can
be a powerful way to understand the class, so property methods are grouped together in
this section. Property accessor methods are listed alphabetically by property name, not by
method name. This means that the "set," "get," and "is" methods for a property all appear
together.

**Public instance methods**
Contains all public instance methods that aren't grouped elsewhere.

**Implementing methods**
Groups together the methods that implement the same interface. There is one subgroup
for each interface implemented by the class. Methods that are defined by the same
interface are almost always related to each other, so this is a useful functional grouping of
methods.

Note that if an interface method is also an event registration method or a property
accessor method, it is listed both in this group and in the event or property group. This
situation does not arise often, but when it does, all of the functional groupings are
important and useful enough to warrant the duplicate listing. When an interface method is
listed in the event or property group, it displays an Implements: flag that specifies the
name of the interface of which it is part.

**Overriding methods**
Groups together the methods that override methods of a superclass, broken down into
subgroups by superclass. This is a useful grouping, because it helps to make clear how a
class modifies the default behavior of its superclasses. Also, in practice, it is often true
that methods that override the same superclass are functionally related to each other.
Sometimes a method that overrides a superclass is also a property accessor method or
(more rarely) an event registration method. When this happens, the method is grouped
with the property or event methods and displays a flag that indicates which superclass it
overrides. The method is not listed with other overriding methods, however. Note that this
is different from interface methods, which, because they are more functionally related, may have duplicate listings in both groups.

**Protected instance methods**
Contains the protected instance methods that aren't grouped elsewhere.

**Fields**
Lists all of the non-constant fields of the class, breaking them down into subgroups for public and protected static and instance fields. Many classes do not define any publicly accessible fields. Many object-oriented programmers prefer not to use publicly accessible fields directly; instead, they use accessor methods when they are available.

**Deprecated members**
Deprecated methods and deprecated fields are grouped at the very bottom of the class synopsis. Using these members is strongly discouraged.

### 8.2.4 Class Hierarchy

For any class or interface that has a nontrivial class hierarchy, the class synopsis is followed by a "Hierarchy" section. This section lists all of the superclasses of the class, as well as any interfaces implemented by those superclasses. It may also list any interfaces extended by an interface. In the hierarchy listing, arrows indicate superclass-to-subclass relationships, while the interfaces implemented by a class follow the class name in parentheses. For example, the quick-reference entry for `net.jxta.protocol.PipeAdvertisement` includes the following hierarchy:

```
Object → net.jxta.protocol.Advertisement → PipeAdvertisement → Cloneable
```

This hierarchy indicates that the `PipeAdvertisement` implements `Cloneable` and extends `net.jxta.document.Advertisement`, which in turn extends `java.lang.Object`.

If a class has subclasses, the "Hierarchy" section is followed by a "Subclasses" section that lists those subclasses. If an interface has implementations, the "Hierarchy" section is followed by an "Implementations" section that lists those implementations. While the "Hierarchy" section shows ancestors of the class, the "Subclasses" and "Implementations" sections show descendants.

### 8.2.5 Cross References

The class hierarchy section of a quick-reference entry is followed by a number of optional "cross-reference" sections that indicate related classes and methods that may be of interest. These sections are the following:

**Passed To**
This section lists all of the methods and constructors that are passed an object of this type as an argument. This is useful when you have an object of a given type and want to figure out what you can do with it.

**Returned By**
This section lists all of the methods (but not the constructors) that return an object of this type. This is useful when you know that you want to work with an object of this type, but you don’t know how to obtain one.

**Thrown By**
For checked exception classes, this section lists all of the methods and constructors that throw exceptions of this type. This material helps you figure out when a given exception or error may be thrown. Note, however, that this section is based on the exception types listed in the `throws` clauses of methods and constructors. Subclasses of `RuntimeException` and `Error` do not have to be listed in throws clauses, so it is not possible to generate a complete cross reference of methods that throw these types of unchecked exceptions.

**Type Of**
This sections lists all of the fields and constants that are of this type, which can help you figure out how to obtain an object of this type.
8.2.6 A Note About Class Names

Throughout the quick reference, you'll notice that classes are sometimes referred to by class name alone, and at other times referred to by class name and package name. If package names were always used, the class synopses would become long and hard to read. On the other hand, if package names were never used, it would be difficult to know which class was being referred to. Package names are omitted for all classes in the java.lang package and for any classes that are in the same package as the class currently being documented.
Bibliography

Colophon

Our look is the result of reader comments, our own experimentation, and feedback from distribution channels. Distinctive covers complement our distinctive approach to technical topics, breathing personality and life into potentially dry subjects.

The animal on the cover of *JXTA in a Nutshell* is a prairie dog. Prairie dogs, named for the barking sound of their cries, are large, bushy rodents that can be found in the prairies and plateaus of the western United States and northern Mexico. They live in burrows that form colonies, or "towns." There are two main species of prairie dogs. The black-tailed variety (*Cynomys ludovicianus*) is more abundant, inhabiting the Great Plains and the Great Basin. This species digs burrows that can be many miles long and include thousands of individuals. The other variety, the white-tailed prairie dog (*Cynomys leucurus*), occurs in higher altitudes than its black-tailed counterpart. While most species of prairie dog are less active in the wintertime, the white-tailed prairie dog hibernates for the entire season. It is also less colonial than *C. ludovicianus*.

Prairie dogs weigh 1 1/2-3 pounds. They are 11-13 inches long, with a tail of 3-4 inches. Their heads are round and wide, and their fur is varying shades of yellow, with darker ears and a whitish underside. They often raise themselves on their haunches and sit upright in rows (a behavior often referred to as "picket pins" in some areas), and can reach a speed of up to 35 miles per hour for short distances. When danger approaches, the prairie dog will let out a warning bark and retreat into their burrows. They eat mostly native plant life, which consists of grasses, roots, weeds, herbs, and blossoms, but will occasionally dine on insects. All of their water is supplied from the food they eat.

The warning calls that prairie dogs use make up one of the most intricate systems of natural animal languages known to scientists. Amazingly, prairie dogs seem to have particular barks that identify different predators, including hawks, owls, ravens, eagles, badgers, coyotes, ferrets, and snakes.

A female prairie dog will give birth to one litter a year, each consisting of 3-5 young. When born, a prairie dog is blind and hairless. At six weeks old, it ventures above ground and is ready for weening. Adult prairie dogs will often relocate and dig new burrows, leaving their young to fend for themselves. Once deprived of the warning system, young prairie dogs are easy prey for predators.

Matt Hutchinson was the production editor and proofreader for *JXTA in a Nutshell*. Sarah Jane Shangraw copyedited the book. David Futato and Colleen Gorman provided quality control. Lucie Haskins wrote the index.

Hanna Dyer designed the cover of this book, based on a series design by Edie Freedman. The cover image is an original illustration created by Lorrie LeJeune. Emma Colby produced the cover layout with QuarkXPress 4.1 using Adobe's ITC Garamond font.

David Futato designed the interior layout. This book was converted to FrameMaker 5.5.6 with a format conversion tool created by Erik Ray, Jason McIntosh, Neil Walls, and Mike Sierra that uses Perl and XML technologies. The text font is Linotype Birka, the heading font is Adobe Myriad Condensed, and the code font is LucasFont's TheSans Mono Condensed. The illustrations that appear in the book were produced by Robert Romano and Jessamyn Read using Macromedia FreeHand 9 and Adobe Photoshop 6. This colophon was written by Matt Hutchinson.

The online edition of this book was created by the Safari production group (John Chodacki, Becki Maisch, and Madeleine Newell) using a set of Frame-to-XML conversion and cleanup tools written and maintained by Erik Ray, Benn Salter, John Chodacki, and Jeff Liggett.