Project JXTA v2.0: Java™ Programmer's Guide

Sun microsystems

THE NETWORK IS THE COMPUTER™

May 2003

Contents

CHAPTER 1

Introduction

JXTA is a set of open, generalized peer-to-peer (P2P) protocols that allow any connected device on the network — from cell phone to PDA, from PC to server — to communicate and collaborate as peers. The JXTA protocols are independent of any programming language, and multiple implementations (called bindings in Project JXTA) exist for different environments. This document specifically discusses the Project JXTA binding on the Java™ 2 Platform, Standard Edition software (J2SE™).

This document is intended for software developers who would like to write and deploy P2P services and applications using the Java programming language and JXTA technology. It provides an introduction to the JXTA technology, describes the Project JXTA network architecture and key concepts, and includes examples and discussion of essential programming constructs using the Project JXTA J2SE platform binding.

Note – This document is based on the *JXTA 2.0 Protocols Specification,* February 27, 2003 (on-line version available at http://spec.jxta.org). It matches the Project JXTA J2SE platform binding stable release of March 2, 2003 (JXTA_2_0_Stable_20030301 03-02-2003).

Why Project JXTA?

As the Web continues to grow in both content and the number of connected devices, peer-to-peer computing is becoming increasingly popular. Popular software based on P2P technologies includes file sharing, distributed computing, and instant messenger services. While each of these applications performs different tasks, they all share many of the same properties, such as discovery of peers, searching, and file or data transfer. Currently, application development is inefficient, with developers solving the same problems and duplicating similar infrastructure implementation. And, most applications are specific to a single platform and are unable to communicate and share data with other applications.

One primary goal of Project JXTA is to provide a platform with the basic functions necessary for a P2P network. In addition, JXTA technology seeks to overcome potential shortcomings in many of the existing P2P systems:

- *Interoperability* — JXTA technology is designed to enable peers providing various P2P services to locate each other and communicate with each other.
- *Platform independence* — JXTA technology is designed to be independent of programming languages, transport protocols, and deployment platforms.

- *Ubiquity* — JXTA technology is designed to be accessible by any device with a digital heartbeat, not just PCs or a specific deployment platform.

One common characteristic of peers in a P2P network is that they often exist on the edge of the regular network. Because they are subject to unpredictable connectivity with potentially variable network addresses, they are outside the standard scope of DNS. JXTA accommodates peers on the edge of the network by providing a system for uniquely addressing peers that is independent of traditional name services. Through the use of JXTA IDs, a peer can wander across networks, changing transports and network addresses, even being temporarily disconnected, and still be addressable by other peers.

What is Project JXTA?

Project JXTA is an open network computing platform designed for peer-to-peer (P2P) computing. Its goal is to develop basic building blocks and services to enable innovative applications for peer groups.

The term "JXTA" is short for juxtapose, as in side by side. It is a recognition that P2P is juxtaposed to client-server or Web-based computing, which is today's traditional distributed computing model.

JXTA provides a common set of open protocols and an open source reference implementation for developing peer-to-peer applications.The JXTA protocols standardize the manner in which peers:

- Discover each other
- Self-organize into peer groups
- Advertise and discover network services
- Communicate with each other
- Monitor each other

The JXTA protocols are designed to be independent of programming languages, and independent of transport protocols. The protocols can be implemented in the Java programming language, C/C++, Perl, and numerous other languages. They can be implemented on top of TCP/IP, HTTP, Bluetooth, HomePNA, or other transport protocols.

What can be done with JXTA Technology?

The JXTA protocols enable developers to build and deploy interoperable P2P services and applications. Because the protocols are independent of both programming language and transport protocols, heterogeneous devices with completely different software stacks can interoperate with one another. Using JXTA technology, developers can write networked, interoperable applications that can:

- Find other peers on the network with dynamic discovery across firewalls
- Easily share documents with anyone across the network
- Find up to the minute content at network sites
- Create a group of peers that provide a service
- Monitor peer activities remotely
- Securely communicate with other peers on the network

Where to get the JXTA technology

Information on JXTA technology can be found at the Project JXTA Web site http://www.jxta.org. This Web site contains project information, developer resources, and documentation. Source code, binaries, documentation, and tutorials are all available for download.

Getting Involved

As with any open source project, a primary goal is to get the community involved by contributing to Project JXTA. Two suggestions for getting started include joining a Project JXTA mailing list and chatting with other JXTA technology enthusiasts.

- Join a mailing list

  Join the mailing lists to post general feedback, feature requests, and requests for help. See the mailing lists page *http://www.jxta.org/maillist.html* for details on how to subscribe.

  Current mailing lists include:

  - *discuss@jxta.org* — topics related to JXTA technology and the community
  - *announce@jxta.org* — Project JXTA announcements and general information
  - *dev@jxta.org* — technical issues for developers
  - *user@jxta.org* — issues for new Project JXTA developers and users

- Chat with other Project JXTA enthusiasts

  You can chat with other Project JXTA users and contributors using the myJXTA demonstration application which can be downloaded at:
  http://download.jxta.org/easyinstall/install.html.

  The demonstration application is available for the following platforms: Microsoft Windows, Solaris™ Operating Environment, Linux, UNIX, MacOS, and other Java technology enabled platforms.

  You can also connect with other Project JXTA users at the #jxta channel on thePeer-Directed Projects Center Internet Relay Chat (IRC) Network:
  http://www.freenode.net/

As you gain experience working with the JXTA technology, you can continue to contribute by filing bug reports, writing or extending tutorials, contributing to existing projects, and proposing new projects.

CHAPTER 2

JXTA Architecture

Overview

The Project JXTA software architecture is divided into three layers, as shown in Figure 2-1.

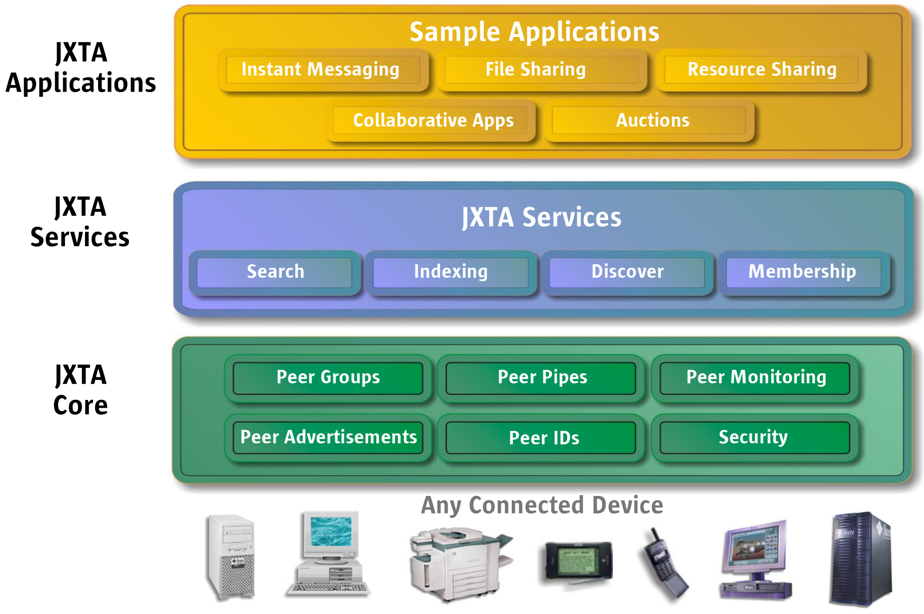


Figure 2-1 Project JXTA software architecture.

- *Platform Layer (JXTA Core)*

  The platform layer, also known as the JXTA core, encapsulates minimal and essential primitives that are common to P2P networking. It includes building blocks to enable key mechanisms for P2P applications, including discovery, transport (including firewall handling), the creation of peers and peer groups, and associated security primitives.

- *Services Layer*

  The services layer includes network services that may not be absolutely necessary for a P2P network to operate, but are common or desirable in the P2P environment. Examples of network services include searching and indexing, directory, storage systems, file sharing, distributed file systems, resource aggregation and renting, protocol translation, authentication, and PKI (Public Key Infrastructure) services.

- *Applications Layer*

  The applications layer includes implementation of integrated applications, such as P2P instant messaging, document and resource sharing, entertainment content management and delivery, P2P E-mail systems, distributed auction systems, and many others.

The boundary between services and applications is not rigid. An application to one customer can be viewed as a service to another customer. The entire system is designed to be modular, allowing developers to pick and choose a collection of services and applications that suits their needs.

JXTA Components

The JXTA network consists of a series of interconnected nodes, or *peers*. Peers can self-organize into *peer groups*, which provide a common set of services. Examples of services that could be provided by a peer group include document sharing or chat applications.

JXTA peers advertise their services in XML documents called *advertisements.* Advertisements enable other peers on the network to learn how to connect to, and interact with, a peer's services.

JXTA peers use *pipes* to send *messages* to one another. Pipes are an asynchronous and unidirectional message transfer mechanism used for service communication. Messages are simple XML documents whose envelope contains routing, digest, and credential information. Pipes are bound to specific *endpoints*, such as a TCP port and associated IP address.

These concepts are described in detail in the following chapters.

Key aspects of the JXTA architecture

Three essential aspects of the JXTA architecture distinguish it from other distributed network models:

- The use of XML documents (advertisements) to describe network resources.
- Abstraction of pipes to peers, and peers to endpoints without reliance upon a central naming/addressing authority such as DNS.
- A uniform peer addressing scheme (peer IDs).

CHAPTER 3

JXTA Concepts

This chapter defines key JXTA terminology and describes the primary components of the JXTA platform.

Peers

A *peer* is any networked device that implements one or more of the JXTA protocols. Peers can include sensors, phones, and PDAs, as well as PCs, servers, and supercomputers. Each peer operates independently and asynchronously from all other peers, and is uniquely identified by a Peer ID.

Peers publish one or more network interfaces for use with the JXTA protocols. Each published interface is advertised as a *peer endpoint,* which uniquely identifies the network interface. Peer endpoints are used by peers to establish direct point-to-point connections between two peers.

Peers are not required to have direct point-to-point network connections between themselves. Intermediary peers may be used to route messages to peers that are separated due to physical network connections or network configuration (e.g., NATS, firewalls, proxies).

Peers are typically configured to spontaneously discover each other on the network to form transient or persistent relationships called peer groups.

Peer Groups

A *peer group* is a collection of peers that have agreed upon a common set of services. Peers self-organize into peer groups, each identified by a unique peer group ID. Each peer group can establish its own membership policy from open (anybody can join) to highly secure and protected (sufficient credentials are required to join).

Peers may belong to more than one peer group simultaneously. By default, the first group that is instantiated is the Net Peer Group. All peers belong to the Net Peer Group. Peers may elect to join additional peer groups.

The JXTA protocols describe how peers may publish, discover, join, and monitor peer groups; they do not dictate when or why peer groups are created.

There are several motivations for creating peer groups:

- *To create a secure environment*

  Groups create a local domain of control in which a specific security policy can be enforced. The security policy may be as simple as a plain text username/password exchange, or as sophisticated as public key cryptography. Peer group boundaries permit member peers to access and publish protected contents. Peer groups form logical regions whose boundaries limit access to the peer group resources.

- *To create a scoping environment*

  Groups allow the establishment of a local domain of specialization. For example, peers may group together to implement a document sharing network or a CPU sharing network. Peer groups serve to subdivide the network into abstract regions providing an implicit scoping mechanism. Peer group boundaries define the search scope when searching for a group's content.

- *To create a monitoring environment*

  Peer groups permit peers to monitor a set of peers for any special purpose (e.g., heartbeat, traffic introspection, or accountability).

Groups also form a hierarchical parent-child relationship, in which each group has single parent. Search requests are propagated within the group. The advertisement for the group is published in the parent group in addition to the group itself.

A peer group provides a set of services called *peer group services.* JXTA defines a core set of peer group services. Additional services can be developed for delivering specific services. In order for two peers to interact via a service, they must both be part of the same peer group.

The core peer group services include the following:

- *Discovery Service* — The discovery service is used by peer members to search for peer group resources, such as peers, peer groups, pipes and services.
- *Membership Service* — The membership service is used by current members to reject or accept a new group membership application. Peers wishing to join a peer group must first locate a current member, and then request to join. The application to join is either rejected or accepted by the collective set of current members. The membership service may enforce a vote of peers or elect a designated group representative to accept or reject new membership applications.
- *Access Service* — The access service is used to validate requests made by one peer to another. The peer receiving the request provides the requesting peers credentials and information about the request being made to determine if the access is permitted. [Note: not all actions within the peer group need to be checked with the access service; only those actions which are limited to some peers need to be checked.]
- *Pipe Service* — The pipe service is used to create and manage pipe connections between the peer group members.
- *Resolver Service* — The resolver service is used to send generic query requests to other peers. Peers can define and exchange queries to find any information that may be needed (e.g., the status of a service or the state of a pipe endpoint).
- *Monitoring Service* — The monitoring service is used to allow one peer to monitor other members of the same peer group.

Not all the above services must be implemented by every peer group. A peer group is free to implement only the services it finds useful, and rely on the default net peer group to provide generic implementations of non-critical core services.

Network Services

Peers cooperate and communicate to publish, discover, and invoke *network services*. Peers can publish multiple services. Peers discover network services via the Peer Discovery Protocol.

The JXTA protocols recognize two levels of network services:

- *Peer Services*

  A peer service is accessible only on the peer that is publishing that service. If that peer should fail, the service also fails. Multiple instances of the service can be run on different peers, but each instance publishes its own advertisement.

- *Peer Group Services*

  A peer group service is composed of a collection of instances (potentially cooperating with each other) of the service running on multiple members of the peer group. If any one peer fails, the collective peer group service is not affected (assuming the service is still available from another peer member). Peer group services are published as part of the peer group advertisement.

Services can be either pre-installed onto a peer or loaded from the network. In order to actually run a service, a peer may have to locate an implementation suitable for the peer's runtime environment.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.

Modules

JXTA modules are an abstraction used to represent any piece of "code" used to implement a behavior in the JXTA world. Network services are the most common example of behavior that can be instantiated on a peer. The module abstraction does not specify what this "code" is: it can be a Java class, a Java jar, a dynamic library DLL, a set of XML messages, or a script. The implementation of the module behavior is left to module implementors. For instance, modules can be used to represent different implementations of a network service on different platforms, such as the Java platform, Microsoft Windows, or the Solaris Operating Environment.

Modules provides a generic abstraction to allow a peer to instantiate a new behavior. As peers browse or join a new peer group, they may find new behaviors that they may want to instantiate. For example, when joining a peer group, a peer may have to learn a new search service that is only used in this peer group. In order to join this group, the peer must instantiate this new search service. The module framework enables the representation and advertisement of platform-independent behaviors, and allows peers to describe and instantiate any type of implementation of a behavior. For example, a peer has the ability to instantiate either a Java or a C implementation of the behavior.

The ability to describe and publish platform-independent behavior is essential to support peer groups composed of heterogeneous peers. The module advertisements enable JXTA peers to describe a behavior in a platform-independent manner. The JXTA platform uses module advertisements to self-describe itself.

The module abstraction includes a module class, module specification, and module implementation:

- *Module Class*

  The module class is primarily used to advertise the existence of a behavior. The class definition represents an expected behavior and an expected binding to support the module. Each module class is identified by a unique ID, the ModuleClassID.

- *Module Specification*

  The module specification is primarily used to access a module. It contains all the information necessary to access or invoke the module. For instance, in the case of a service, the module specification may contain a pipe advertisement to be used to communicate with the service.

  A module specification is one approach to providing the functionality that a module class implies. There can be multiple module specifications for a given module class. Each module specification is identified by a unique ID, the ModuleSpecID. The ModuleSpecID contains the ModuleClass ID (i.e., the ModuleClassID is embedded in a ModuleSpecID), indicating the associated module class.

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

- *Module Implementation*

  The module implementation is the implementation of a given module specification. There may be multiple module implementations for a given module specification. Each module implementation contains the ModuleSpecID of the associated specification it implements.

Modules are used by peer groups services, and can also be used by stand-alone services. JXTA services can use the module abstraction to identify the existence of the service (its Module Class), the specification of the service (its Module Specification), or an implementation of the service (a Module Implementation). Each of these components has an associated advertisement, which can be published and discovered by other JXTA peers.

As an example, consider the JXTA Discovery Service. It has a unique ModuleClassID, 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 may use different strategies tailored to the size of the group and its dispersion across the network, while another experiments with new strategies. Each specification has a unique ModuleSpecID, which references the discovery service ModuleClassID. For each specification, there can be multiple implementations, each of which contains the same ModuleSpecID.

In summary, there can be multiple specifications of a given module class, and each may be incompatible. However, all implementations of any given specification are assumed to be compatible.

Pipes

JXTA peers use *pipes* to send messages to one another. Pipes are an asynchronous and unidirectional message transfer mechanism used for service communication. Pipes are indiscriminate; they support the transfer of any object, including binary code, data strings, and Java technology-based objects.

The pipe endpoints are referred to as the *input pipe* (the receiving end) and the *output pipe* (the sending end). Pipe endpoints are dynamically bound to peer endpoints at runtime. Peer endpoints correspond to available peer network interfaces (e.g., a TCP port and associated IP address) that can be used to send and receive message. JXTA pipes can have endpoints that are connected to different peers at different times, or may not be connected at all.

Pipes are virtual communication channels and may connect peers that do not have a direct physical link. In this case, one or more intermediary peer endpoints are used to relay messages between the two pipe endpoints.

Pipes offer two modes of communication, point-to-point and propagate, as seen in Figure 3-1. The JXTA core also provides secure unicast pipes, a secure variant of the point-to-point pipe.

- *Point-to-point Pipes*

  A point-to-point pipe connects exactly two pipe endpoints together: an input pipe on one peer receives messages sent from the output pipe of another peer.

- *Propagate Pipes*

  A propagate pipe connects one output pipe to multiple input pipes. Messages flow from the output pipe (the propagation source) into the input pipes. All propagation is done within the scope of a peer group. That is, the output pipe and all input pipes must belong to the same peer group.

- *Secure Unicast Pipes*

  A secure unicast pipe is a type of point-to-point pipe that provides a secure communication channel.

Additional types of pipe services can be built using the basic core pipes. For example, the current J2SE platform binding includes bidirectional pipes and bidirectional/reliable pipes.

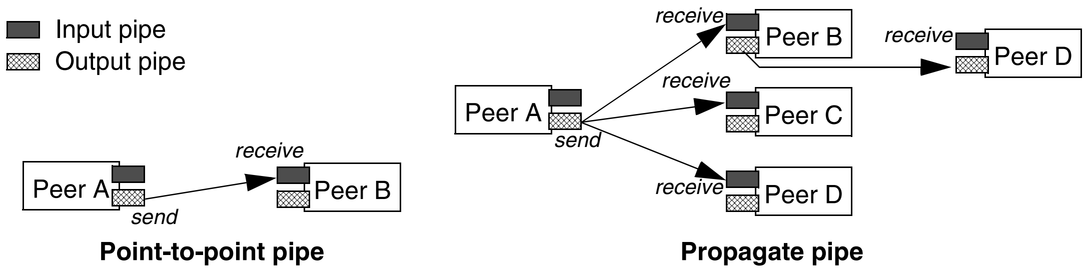


Figure 3-1 Point-to-point and propagate pipes.

Messages

A message is an object that is sent between JXTA peers; it is the basic unit of data exchange between peers. Messages are sent and received by the Pipe Service and by the Endpoint Service. Typically, applications use the Pipe Service to create, send, and receive messages. (In general, applications are not expected to need to use the Endpoint Service directly. If, however, an application needs to understand or control the topology of the JXTA network, the Endpoint Service can be used.)

A message is an ordered sequence of named and typed contents called message elements. Thus a message is essentially a set of name/value pairs. The content can be an arbitrary type.

The JXTA protocols are specified as a set of messages exchanged between peers. Each software platform binding describes how a message is converted to and from a native data structure such as a Java technology object or a C structure.

There are two representations for messages: XML and binary. The JXTA J2SE platform binding uses a binary format envelop to encapsulate the message payload. Services can use the most appropriate format for that transport (e.g., a service which requires a compact representation for a messages can use the binary representation, while other services can use XML). Binary data may be encoded using a Base64 encoding scheme in the body of an XML message.

The use of XML messages to define protocols allows many different kinds of peers to participate in a protocol. Because the data is tagged, each peer is free to implement the protocol in a manner best-suited to its abilities and role. If a peer only needs some subset of the message, the XML data tags enable that peer to identify the parts of the message that are of interest. For example, a peer that is highly constrained and has insufficient capacity to process some or most of a message can use data tags to extract the parts that it can process, and can ignore the remainder.

Advertisements

All JXTA network resources — such as peers, peer groups, pipes, and services — are represented by an *advertisement*. Advertisements are language-neutral metadata structures represented as XML documents. The JXTA protocols use advertisements to describe and publish the existence of a peer resources. Peers discover resources by searching for their corresponding advertisements, and may cache any discovered advertisements locally.

Each advertisement is published with a lifetime that specifies the availability of its associated resource. Lifetimes enable the deletion of obsolete resources without requiring any centralized control. An advertisement can be republished (before the original advertisement expires) to extend the lifetime of a resource.

The JXTA protocols define the following advertisement types:

- *Peer Advertisement* — describes the peer resource. The primary use of this advertisement is to hold specific information about the peer, such as its name, peer ID, available endpoints, and any run-time attributes which individual group services want to publish (such as being a rendezvous peer for the group).
- *Peer Group Advertisement* — describes peer group-specific resources, such as name, peer group ID, description, specification, and service parameters.
- *Pipe Advertisement* — describes a pipe communication channel, and is used by the pipe service to create the associated input and output pipe endpoints. Each pipe advertisement contains an optional symbolic ID, a pipe type (point-to-point, propagate, secure, etc.) and a unique pipe ID.
- *Module Class Advertisement* — describes a module class. Its primary purpose is to formally document the existence of a module class. It includes a name, description, and a unique ID (ModuleClassID).
- *Module Spec Advertisement* — defines a module specification. Its main purpose is to provide references to the documentation needed in order to create conforming implementations of that specification. A secondary use is, optionally, to make running instances usable remotely, by publishing information such as a pipe advertisement. It includes name, description, unique ID (ModuleSpecID), pipe advertisement, and parameter field containing arbitrary parameters to be interpreted by each implementation.
- *Module Impl Advertisement* — defines an implementation of a given module specification. It includes name, associated ModuleSpecID, as well as code, package, and parameter fields which enable a peer to retrieve data necessary to execute the implementation.
- *Rendezvous Advertisement* — describes a peer that acts as a rendezvous peer for a given peer group.
- *Peer Info Advertisement* — describes the peer info resource. The primary use of this advertisement is to hold specific information about the current state of a peer, such as uptime, inbound and outbound message count, time last message received, and time last message sent.

Each advertisement is represented by an XML document. Advertisements are composed of a series of hierarchically arranged elements. Each element can contain its data or additional elements. An element can also have attributes. Attributes are name-value string pairs. An attribute is used to store meta-data, which helps to describe the data within the element.

An example of a pipe advertisement is included in Figure 3-2.

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

<!DOCTYPE jxta:PipeAdvertisement>

<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
       <Id>
          urn:jxta:uuid-
59616261646162614E504720503250338E3E786229EA460DADC1A176B69B731504
       </Id>
       <Type>
          JxtaUnicast
       </Type>
       <Name>
          TestPipe.end1
       </Name>
</jxta:PipeAdvertisement>
```

Figure 3-2 Example JXTA pipe advertisement.

The complete specification of the JXTA advertisements is given in the *JXTA Protocols Specification.* Services or peer implementations may subtype any of the above advertisements to create their own advertisements.

Security

Dynamic P2P networks such as the JXTA network need to support different levels of resource access. JXTA peers operate in a role-based trust model, in which an individual peer acts under the authority granted to it by another trusted peer to perform a particular task.

Five basic security requirements must be provided:

- *Confidentiality* — guarantees that the contents of a message are not disclosed to unauthorized individuals.
- *Authentication* — guarantees that the sender is who he or she claims to be.
- *Authorization* — guarantees that the sender is authorized to send a message.
- *Data integrity* — guarantees that the message was not modified accidentally or deliberately in transit.
- *Refutability* — guarantees that the message was transmitted by a properly identified sender and is not a replay of a previously transmitted message.

XML messages provide the ability to add metadata such as credentials, certificates, digests, and public keys to JXTA messages, enabling these basic security requirements to be met. Message digests guarantee the data integrity of messages. Messages may also be encrypted (using public keys) and signed (using certificates) for confidentiality and refutability. Credentials can be used to provide message authentication and authorization.

A credential is a token that is used to identify a sender, and can be used to verify a sender's right to send a message to a specified endpoint. The credential is an opaque token that must be presented each time a message is sent.The sending address placed in a JXTA message envelope is cross-checked with the sender's identity in the credential. Each credential's implementation is specified as a plug-in configuration, which allows multiple authentication configurations to co-exists on the same network.

It is the intent of the JXTA protocols to be compatible with widely accepted transport-layer security mechanisms for message-based architectures, such as Secure Sockets Layer (SSL) and Internet Protocol Security (IPSec). However, secure transport protocols such as SSL and IPSec only provide the integrity and confidentiality of message transfer between two communicating peers. In order to provide secure transfer in a multi-hop network like JXTA, a trust association must be established among all intermediary peers. Security is compromised if any one of the communication links is not secured.

IDs

Peers, peer groups, pipes and other JXTA resources need to be uniquely identifiable. A JXTA ID uniquely identifies an entity and serves as a canonical way of referring to that entity. Currently, there are six types of JXTA entities which have JXTA ID types defined: peers, peer group, pipes, contents, module classes, and module specifications.

URNs are used to express JXTA IDs. URNs[1] are a form of URI that "... are intended to serve as persistent, location-independent, resource identifiers". Like other forms of URI, JXTA IDs are presented as text.

An example JXTA peer ID is:

```
    urn:jxta:uuid-59616261646162614A78746150325033F3BC76FF13C2414CBC0AB663666DA53903
```

An example JXTA pipe ID is:

```
    urn:jxta:uuid-59616261646162614E504720503250338E3E786229EA460DADC1A176B69B731504
```

Unique IDs are generated randomly by the JXTA J2SE platform binding. There are two special reserved JXTA IDs: the NULL ID and the Net Peer Group ID.

1. See IETF RFC 2141 for more information on URNs.

CHAPTER 4

Network Architecture

Network Organization

The JXTA network is an ad hoc, multi-hop, and adaptive network composed of connected peers. Connections in the network may be transient, and message routing between peers is nondeterministic. Peers may join or leave the network at any time, and routes may change frequently.

Peers may take any form as long as they can communicate using JXTA protocols. The organization of the network is not mandated by the JXTA framework, but in practice four kinds of peers are typically used:

- *Minimal edge peer*

  A minimal edge peer can send and receive messages, but does not cache advertisements or route messages for other peers. Peers on devices with limited resources (e.g., a PDA or cell phone) would likely be minimal edge peers.

- *Full-featured edge peer*

  A full-featured peer can send and receive messages, and will typically cache advertisements. A simple peer replies to discovery requests with information found in its cached advertisements, but does not forward any discovery requests. Most peers are likely to be edge peers.

- *Rendezvous peer*

  A rendezvous peer is like any other peer, and maintains a cache of advertisements. However, rendezvous peers also forward discovery requests to help other peers discover resources. When a peer joins a peer group, it automatically seeks a rendezvous peer.[1] If no rendezvous peer is found, it dynamically becomes a rendezvous peer for that peer group. Each rendezvous peer maintains a list of other known rendezvous peers and also the peers that are using it as a rendezvous.

  Each peer group maintains its own set of rendezvous peers, and may have as many rendezvous peers as needed. Only rendezvous peers that are a member of a peer group will see peer group specific search requests.

1. In the JXTA 2.0 release, a peer will be connected to at most one rendezvous peer at any given time.

Edge peers send search and discovery requests to rendezvous peers, which in turn forward requests they cannot answer to other known rendezvous peers. The discovery process continues until one peer has the answer or the request dies. Messages have a default time-to-live (TTL) of seven hops. Loopbacks are prevented by maintaining the list of peers along the message path.

- *Relay peer*[2]

  A relay peer maintains information about the routes to other peers and routes messages to peers. A peer first looks in its local cache for route information. If it isn't found, the peer sends queries to relay peers asking for route information. Relay peers also forward messages on the behalf of peers that cannot directly address another peer (e.g., NAT environments), bridging different physical and/or logical networks

Any peer can implement the services required to be a relay or rendezvous peer. The relay and rendezvous services can be implemented as a pair on the same peer.

Shared Resource Distributed Index

The JXTA 2.0 J2SE platform supports a shared resource distributed index (SRDI) service to provide a more efficient mechanism for propagating query requests within the JXTA network. Rendezvous peers maintain an index of advertisements published by edge peers. When edge peers publish new advertisements, they use the SRDI service to push advertisement indices to their rendezvous. With this rendezvous-edge peer hierarchy, queries are propagated between rendezvous only, which significantly reduces the number of peers involved in the search for an advertisement.

Each rendezvous maintains its own list of known rendezvous in the peer group. A rendezvous may retrieve rendezvous information from a pre-defined set of bootstrapping, or seeding, rendezvous. Rendezvous periodically select a given random number of rendezvous peers and send them a random list of their known rendezvous. Rendezvous also periodically purge non-responding rendezvous. Thus, they maintain a loosely-consistent network of known rendezvous peers.

When a peer publishes a new advertisement, the advertisement is indexed by the SRDI service using keys such as the advertisement name or ID. Only the indices of the advertisement are pushed to the rendezvous by SRDI, minimizing the amount of data that needs to be stored on the rendezvous. The rendezvous also pushes the index to additional rendezvous peers (selected by the calculation of a hash function of the advertisement index). [3]

2. Relay peers were referred to as router peers in earlier documentation.

3. See *Project JXTA: A Loosely-Consistent DHT Rendezvous Walker*, a technical white paper by Bernard Traversat, Mohamed Abdelaziz, and Eric Pouyoul, for more detailed information on the implementation.

Queries

An example configuration is shown in Figure 4-1. Peer A is an edge peer, and is configured to use Peer R1 as its rendezvous. When Peer A initiates a discovery or search request, it is initially sent to its rendezvous peer — R1, in this example — and also via multicast to other peers on the same subnet.

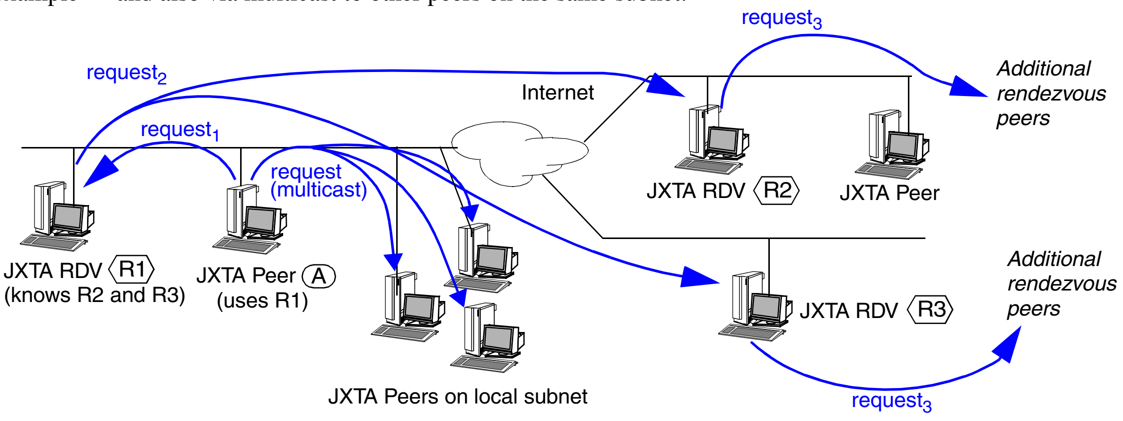


Figure 4-1 Request propagation via rendezvous peers.

Local neighborhood queries (i.e., within a subnet) are propagated to neighboring peers using what a transport defines as the broadcast or multicast method. Peers receiving the query respond directly to the requesting peer, if they contain the information in their local cache.

Queries beyond the local neighborhood are sent to the connected rendezvous peer. The rendezvous peer attempts to satisfy the query against its local cache. If it contains the requested information, it replies directly to the requesting peer and does not further propagate the request. If it contains the index for the resource in its SRDI, it will notify the peer that published the resource and that peer will respond directly to the requesting peer. (Recall that the rendezvous stores only the index for the advertisement, and not the advertisement itself.)

If the rendezvous peer does not contain the requested information, a default limited-range walker algorithm is used to walk the set of rendezvous looking for a rendezvous that contains the index. A hop count is used to specify the maximum number of times the request can be forwarded. Once the query reaches the peer, it replies directly to the originator of the query.

Figure 4-2 depicts a logical view of how the SRDI service works. Peer 2 publishes a new advertisement, and a SRDI message is sent to its rendezvous, R3. Indices will be stored on R3, and may be pushed to other rendezvous in the peer group. Now, Peer 1 sends a query request for this resource to its rendezvous, R1. Rendezvous R1 will check its local cache of SRDI entries, and will propagate the query if it is not found. When the resource is located on Peer 2, Peer 2 will respond directly to P1 with the requested advertisement.

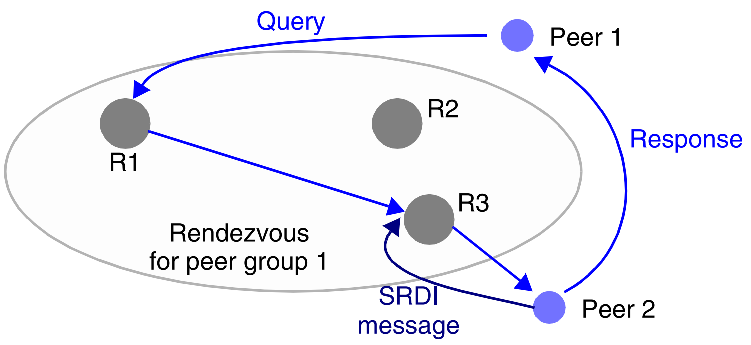


Figure 4-2 Distributing and querying for SRDI resource entries.

Firewalls and NAT

A peer behind a firewall can send a message directly to a peer outside a firewall. But a peer outside the firewall cannot establish a connection directly with a peer behind the firewall.

In order for JXTA peers to communicate with each other across a firewall, the following conditions must exist:

- At least one peer in the peer group inside the firewall must be aware of at least one peer outside of the firewall.
- The peer inside and the peer outside the firewall must be aware of each other and must support HTTP.
- The firewall has to allow HTTP data transfers.

These HTTP transfers across the firewall need not be restricted to port 80, although that is the port used by default by most Web browsers and by the current Project JXTA J2SE binding.

Figure 4-3 depicts a typical message routing scenario through a firewall. In this scenario, JXTA Peers A and B want to pass a message, but the firewall prevents them from communicating directly. JXTA Peer A first makes a connection to Peer C using a protocol such as HTTP that can penetrate the firewall. Peer C then makes a connection to Peer B, using a protocol such as TCP/IP. A virtual connection is now made between Peers A and B.

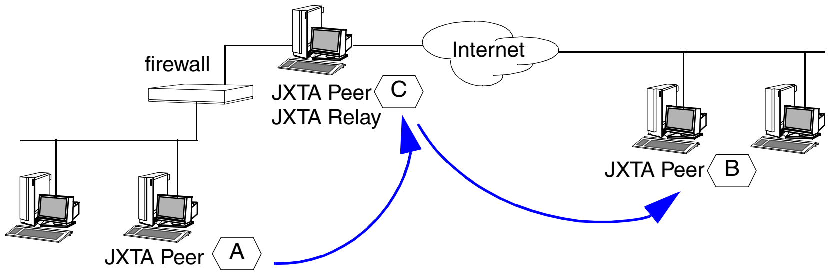


Figure 4-3 Message routing scenario across a firewall.

CHAPTER 5

JXTA Protocols

JXTA defines a series of XML message formats, or *protocols*, for communication between peers. Peers use these protocols to discover each other, advertise and discover network resources, and communication and route messages.

There are six JXTA protocols:

- *Peer Discovery Protocol (PDP)* — used by peers to advertise their own resources (e.g., peers, peer groups, pipes, or services) and discover resources from other peers. Each peer resource is described and published using an advertisement.
- *Peer Information Protocol (PIP)* — used by peers to obtain status information (uptime, state, recent traffic, etc.) from other peers.
- *Peer Resolver Protocol (PRP)* **—** enables peers to send a generic query to one or more peers and receive a response (or multiple responses) to the query. Queries can be directed to all peers in a peer group or to specific peers within the group. Unlike PDP and PIP, which are used to query specific pre-defined information, this protocol allows peer services to define and exchange any arbitrary information they need.
- *Pipe Binding Protocol (PBP)* — used by peers to establish a virtual communication channel, or *pipe*, between one or more peers. The PBP is used by a peer to bind two or more ends of the connection (pipe endpoints).
- *Endpoint Routing Protocol (ERP)* — used by peers to find routes (paths) to destination ports on other peers. Route information includes an ordered sequence of relay peer IDs that can be used to send a message to the destination. (For example, the message can be delivered by sending it to Peer A which relays it to Peer B which relays it to the final destination.)
- *Rendezvous Protocol (RVP)* — mechanism by which peers can subscribe or be a subscriber to a propagation service. Within a peer group, peers can be rendezvous peers or peers that are listening to rendezvous peers. the RVP allows a peer to send messages to all listening instances of the service. The RVP is used by the Peer Resolver Protocol and the Pipe Binding Protocol to propagate messages.

All JXTA protocols are asynchronous, and are based on a query/response model. A JXTA peer uses one of the protocols to send a query to one or more peers in its peer group. It may receive zero, one, or more responses to its query. For example, a peer may use PDP to send a discovery query asking for all known peers in the default Net Peer Group. In this case, multiple peers will likely reply with discovery responses. In another example, a peer may send a discovery request asking for a specific pipe named “aardvark”. If this pipe isn’t found, then zero discovery responses will be sent in reply.

JXTA peers are not required to implement all six protocols; they only need implement the protocols they will use. The current Project JXTA J2SE platform binding supports all six JXTA protocols. The Java programming language API is used to access operations supported by these protocols, such as discovering peers or joining a peer group.

Note – For a complete description of the JXTA protocols, please see the *JXTA Protocols Specification*, available for download from http://spec.jxta.org. This document is based on Revision 1.2.11 of the specification.

Peer Discovery Protocol

The Peer Discovery Protocol (PDP) is used to discover any published peer resources. Resources are represented as advertisements. A resource can be a peer, peer group, pipe, service, or any other resource that has an advertisement.

PDP enables a peer to find advertisements on other peers. The PDP is the default discovery protocol for all user defined peer groups and the default net peer group. Custom discovery services may choose to leverage the PDP. If a peer group does not have its own discovery service, the PDP is used to probe peers for advertisements.

There are multiple ways to discover distributed information. The current Project JXTA J2SE platform binding uses a combination of IP multicast to the local subnet and the use of rendezvous peers, a technique based on network-crawling. Rendezvous peers provide the mechanism of sending requests from one known peer to the next ("crawling" around the network) to dynamically discover information. A peer may be pre-configured with a pre-defined set of rendezvous peers. A peer may also choose to bootstrap itself by dynamically locating rendezvous peers or network resources in its proximity environment. Other techniques, such as content-addressable networks (CANs), could also be added to perform resource discovery.

Peers generate discovery query request messages to discover advertisements within a peer group. This message contains the peer group credential of the probing peer and identifies the probing peer to the message recipient. Messages can be sent to any peer within a region or to a rendezvous peer.

A peer may receive zero, one, or more responses to a discovery query request. The response message returns one or more advertisements.

Peer Information Protocol

Once a peer is located, it capabilities and status may be queried. The Peer Information Protocol (PIP) provides a set of messages to obtain peer status information. This information can be used for commercial or internal deployment of JXTA applications. For example, in commercial deployments the information can be used to determine the usage of a peer service and bill the service consumers for their use. In an internal IT deployment, the information can be used by the IT department to monitor a node's behavior and reroute network traffic to improve overall performance. These hooks can be extended to provide the IT department control of the peer node in addition to providing status information.

The PIP ping message is sent to a peer to check if the peer is alive and to get information about the peer. The ping message specifies whether a full response (peer advertisement) or a simple acknowledgment (alive and uptime) should be returned.

The peerinfo message is used to send a message in response to a ping message. It contains the credential of the sender, the source peer ID and target peer ID, uptime, and peer advertisement.

Peer Resolver Protocol

The Peer Resolver Protocol (PRP) enables peers to send generic query requests to other peers and identify matching responses. Query requests can be sent to a specific peer, or can be propagated via the rendezvous services within the scope of a peer group. The PRP uses the Rendezvous Service to disseminate a query to multiple peers, and uses unicast messages to send queries to specified peers.

The PRP is a foundation protocol supporting generic query requests. Both PIP and PDP are built using PRP, and provide specific query/requests: the PIP is used to query specific status information and PDP is used to discover peer resources. The PRP can be used for any generic query that may be needed for an application. For example, the PRP enables peers to define and exchange queries to find or search service information such as the state of the service, the state of a pipe endpoint, etc.

The resolver query message is used to send a resolver query request to a service on another member of a peer group. The resolver query message contains the credential of the sender, a unique query ID, a specific service handler, and the query. Each service can register a handler in the peer group resolver service to process resolver query requests and generate replies. The resolver response message is used to send a message in response to a resolver query message. The resolver response message contains the credential of the sender, a unique query ID, a specific service handler and the response. Multiple resolver query messages may be sent. A peer may receive zero, one, or more responses to a query request.

Peers may also participate in the Shared Resource Distributed Index (SRDI). SRDI provides a generic mechanism, where JXTA services can utilize a distributed index of shared resources with other peers that are grouped as a set of more capable peers such as rendezvous peers. These indices can be used to direct queries in the direction where the query is most likely to be answered, and repropagate messages to peers interested in these propagated messages.The PRP sends a resolver SRDI message to the named handler on one or more peers in the peer group. The resolver SRDI message is sent to a specific handler, and it contains a string that will be interpreted by the targeted handler.

Pipe Binding Protocol

The Pipe Binding Protocol (PBP) is used by peer group members to bind a pipe advertisement to a pipe endpoint. The pipe virtual link (pathway) can be layered upon any number of physical network transport links such as TCP/IP. Each end of the pipe works to maintain the virtual link and to re-establish it, if necessary, by binding or finding the pipe's currently bound endpoints.

A pipe can be viewed as an abstract named message queue, supporting create, open/resolve (bind), close (unbind), delete, send, and receive operations. Actual pipe implementations may differ, but all compliant implementations use PBP to bind the pipe to an endpoint. During the abstract create operation, a local peer binds a pipe endpoint to a pipe transport.

The PBP query message is sent by a peer pipe endpoint to find a pipe endpoint bound to the same pipe advertisement. The query message may ask for information not obtained from the cache. This is used to obtain the most up-to-date information from a peer. The query message can also contain an optional peer ID, which if present indicates that only the specified peer should respond to the query.

The PBP answer message is sent back to the requesting peer by each peer bound to the pipe. The message contains the Pipe ID, the peer where a corresponding InputPipe has been created, and a boolean value indicating whether the InputPipe exists on the specified peer.

Endpoint Routing Protocol

The Endpoint Routing Protocol (ERP) defines a set of request/query messages that are used to find routing information. This route information is needed to send a message from one peer (the source) to another (the destination). When a peer is asked to send a message to a given peer endpoint address, it first looks in its local cache to determine if it has a route to this peer. If it does not find a route, it sends a route resolver query request to its available peer relays asking for route information. When a peer relay receives a route query, it checks if knows the route. If it does, it returns the route information as an enumeration of hops.

Any peer can query a peer relay for route information, and any peer in a peer group may become a relay. Peer relays typically cache route information.

Route information includes the peer ID of the source, the peer ID of the destination, a time-to-live (TTL) for the route, and an ordered sequence of gateway peer IDs. The sequence of peer IDs may not be complete, but should contain at least the first relay.

Route query requests are sent by a peer to a peer relay to request route information. The query may indicate a preference to bypass the cache content of the router and search dynamically for a new route.

Route answer messages are sent by a relay peer in response to a route information requests. This message contains the peer ID of the destination, the peer ID and peer advertisement of the router that knows a route to the destination, and an ordered sequence of one or more relays.

Rendezvous Protocol

The Rendezvous Protocol (RVP) is responsible for propagating messages within a peer group. While different peer groups may have different means to propagate messages, the Rendezvous Protocol defines a simple protocol that allows:

- Peers to connect to service (be able to propagate messages and receive propagates messages)
- Control the propagation of the message (TTL, loopback detection, etc.).

The RVP is used by the Peer Resolver Protocol and by the Pipe Binding Protocol in order to propagate messages.

CHAPTER 6

Hello World Example

This chapter discusses the steps required to run a simple "Hello World" example, including:

- System requirements
- Accessing the on-line documentation
- Downloading the Project JXTA binaries
- Compiling JXTA technology code
- Running JXTA technology applications
- Configuring the JXTA environment

Getting Started

System Requirements

The current Project JXTA J2SE platform binding requires a platform that supports the Java Run-Time Environment (JRE) or Software Development Kit (SDK) 1.3.1 release or later. This environment is currently available on the Solaris Operating Environment, Microsoft Windows 95/98/2000/ME/NT 4.0, Linux, and Macintosh.

The J2SE platform JRE and SDK for Solaris SPARC/x86, Linux x86, and Microsoft Windows can be downloaded from:

http://java.sun.com/j2se/downloads.html

Accessing On-line Documentation

On-line documentation for the Project JXTA source code is available using Javadoc software at:

http://platform.jxta.org/java/api/overview-summary.html

Downloading Binaries

The required J2SE platform binaries (Java programming language `.jar` files) are listed in Table 6-1.

These binaries can be downloaded from the *http://www.jxta.org* Web site. Use the *Downloads* link to navigate to the page describing the download options.

Name	Description
`jxta.jar`	Project JXTA P2P platform infrastructure building blocks and protocols.
`jxtasecurity.jar`	Used for the security of local storage in the personal security environment. This includes securing both the password and passphrase for the RSA private keys.
`log4j.jar`	Java programming language application logging system.
`jxtaptls.jar`	Used for both certificate management and TLS; the pureTLS implementation includes software developed by Claymore Systems, Inc.
`minimalBC.jar`	Minimalized Bouncy Castle security library; used for certificate creation along with the `jxtaptls.jar`.
`cryptix-asn1.jar`	Required by pureTLS for asn1.
`cryptix32.jar`	Required by pureTLS for the TLS ciphersuites.

Table 6-1 Required Java `.jar` files.

The quickest option is to download one of the Project JXTA builds. There are two types of project builds available:

- *Stable builds* — the most recently saved stable build of the software; these are the best choice for new JXTA users. Easy to use installers for these builds are available by following the link on the Web page to the Project JXTA Easy Installers (*http://download.jxta.org/easyinstall/install.html*). These installers provide an easy way to download JXTA only (if you already have the JVM) or download both JXTA and JVM in one convenient step. The Easy installers install the JXTA demos, JXTA Shell and InstantP2P.
- *Daily builds* — the automated builds of the current "work in progress"; these builds are provided for developer testing, and are not guaranteed to function correctly.

You can also download the Project JXTA source code, and compile the various `.jar` files yourself. Follow the directions on the Project JXTA Web page to download the source code and then build the binaries.

Compiling JXTA Code

The application in this example, SimpleJxtaApp, requires the `jxta.jar` file for compilation. When you run the Java compiler (`javac`[1]), you need to include the `-classpath` option specifying the location of this `.jar` file. For example, users on the Window systems could use a command similar to Figure 6-1, substituting the actual location of the `jxta.jar` file on their system:

```
C:> javac -classpath .\lib\jxta.jar SimpleJxtaApp.java
```

Figure 6-1 Example compilation command (Windows systems).

1. Refer to your documentation for specific details on running the Java programming language compiler on your platform. Some compilers use the `-cp` option to specify the classpath. Alternatively, you may choose to set the `CLASSPATH` environment variable, which has the same effect as specifying the `-classpath` compilation option.

Running JXTA Applications

When you enter the `java`[1] command to run the application, you need to include the `-classpath` option specifying the location of the required `.jar` files (see Table 6-1). For example, users on Window systems could use a command similar to Figure 6-2, substituting the actual location of their `.jar` files:

```
C:> java -classpath .\lib\jxta.jar;.\lib\log4j.jar;
    .\lib\jxtasecurity.jar;.\lib\cryptix-asn1.jar;.\lib\cryptix32.jar;
    .\lib\jxtaptls.jar;.\lib\minimalBC.jar;. SimpleJxtaApp
```

Figure 6-2 Example command to run application (Windows systems).

Note – You may find it easiest to create a script or batch file containing the command to run your application. This eliminates the need to type lengthy commands each time you want to run your application.

Configuration

The first time a JXTA technology application is run, an auto-configuration tool (JXTA Configurator) is displayed to configure the JXTA platform for your network environment. This tool is used to specify configuration information for TCP/IP and HTTP, configure rendezvous and relay peers, and enter a user name and password.

When the JXTA Configurator starts, it displays the Basic Settings panel (see Figure 6-3). Additional panels are displayed by selecting the tabs (Advanced, Rendezvous/Relay, Security) at the top of the panel.

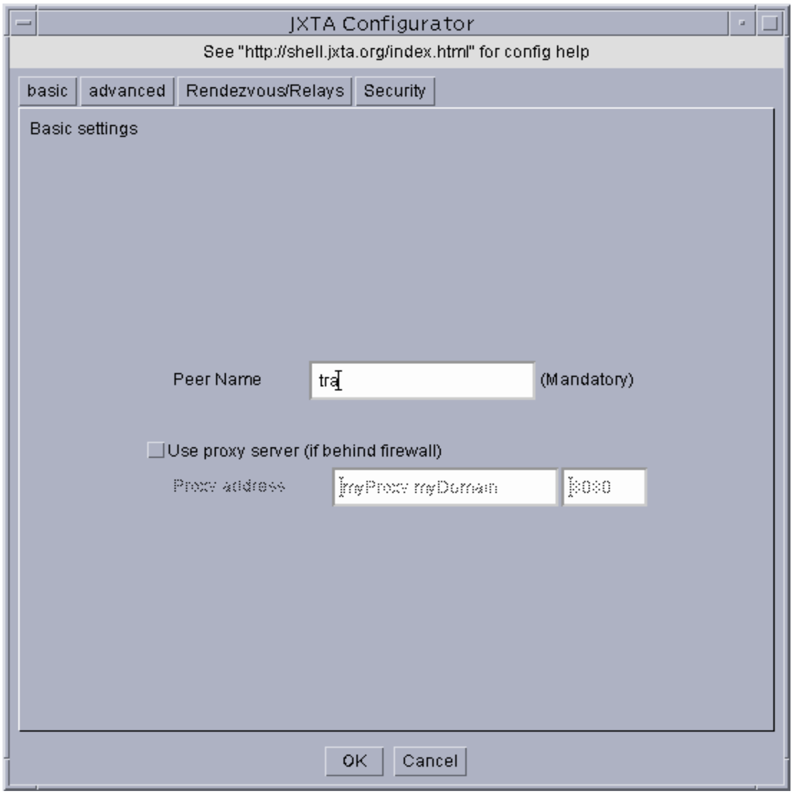
- Basic: You can use any string for your peer name. If your peer is located behind a firewall, you will also need to check the box "Use a proxy server" and enter your proxy server name and port number[2].
- Advanced: This panel is used to specify TCP and HTTP settings. Outgoing TCP connections should be enabled for most situations. If you are not behind a firewall or NAT, incoming TCP connections should also be enabled, and you do not need to use a relay. If you are behind a firewall or NAT, incoming connections should be disabled and a relay is needed in order to communicate.
- Rendezvous/Relay: Download the list of rendezvous and relay peers. If you are behind a firewall or NAT, select Use a Relay.
- Security: Enter a username and password.

Note – For more detailed information on using the JXTA Configurator, please see *http://platform.jxta.org/java/confighelp.html* .

---

1. Again, see your Java documentation for specific details on running applications on your platform. Some environments use the `-cp` option to specify the classpath. Alternatively, you may choose to set the `CLASSPATH` environment variable, which has the same effect as specifying the `-classpath` command line option.

2. If you are not sure if your peer is located behind a firewall, or if you need to determine the correct proxy server to use, please see your network administrator.


JXTA Configurator

See "http://shell.jxta.org/index.html" for config help

Basic settings

Peer Name: tra (Mandatory)

Use proxy server (if behind firewall)

Proxy address: myProxy.myDomain 8080

Figure 6-3 JXTA Configurator: Basic settings.

Configuration information is stored in the file `./.jxta/PlatformConfig`; security information (username and password) is stored in the `./.jxta/pse` subdirectory. The next time the application runs, this information is used to configure your peer. If you would like to re-run the auto-configuration tool, create a file named `reconf` in the `./.jxta` directory. If this file exists when you start your JXTA application, the JXTA Configurator will run and prompt you for new configuration information. (You can also remove the `PlatformConfig` file and then start your application again; The JXTA Configurator runs if there is no `PlatformConfig` file.)

Note – To specifiy an alternate location for the configuration information (rather than using the default `./.jxta` subdirectory), use:

```
java -DJXTA_HOME="alternate dir"
```

HelloWorld Example

This example illustrates how an application can start the JXTA platform. The application instantiates the JXTA platform and then prints a message displaying the peer group name, peer group ID, peer name, and peer ID. Figure 6-4 shows example output when this application is run:

```
Starting JXTA ....
Hello from JXTA group NetPeerGroup
  Group ID = urn:jxta:jxta-NetGroup
  Peer name = suzi
  Peer ID = urn:jxta:uuid-59616261646162614A78746150325033F3B
C76FF13C2414CBC0AB663666DA53903
```

Figure 6-4 Example output: SimpleJxtaApp.

Hello World Example: SimpleJxtaApp

The code for this example begins on page 30. We define a single class, SimpleJxtaApp, with one class variable:

- `PeerGroup netPeerGroup` — our peer group (the default net peer group)

and two methods:

- `static public void main()` — main routine; prints peer and peer group information
- `public void startJxta()` — initializes the JXTA platform and creates the net peer group

startJxta()

The startJxta() method uses a single call to instantiate the JXTA platform [line 34]:

```
netPeerGroup = PeerGroupFactory.newNetPeerGroup();
```

This call instantiates the default platform object and then creates and returns a PeerGroup object containing the default net peer group. This object contains the default reference implementations of the various JXTA services (DiscoveryService, MembershipService, RendezvousService, etc.). It also contains the peer group ID and peer group name, as well as the name and ID of the peer on which we're running.

main()

This method first calls startJxta() to instantiate the JXTA platform. Next, this method prints out various information from our netPeerGroup:

- *Group name* — the name of the default net group, NetPeerGroup [line 18]:
  ```
  System.out.println ("Hello from JXTA group " +
                      netPeerGroup.getPeerGroupName() );
  ```
- *Peer Group ID* — the peer group ID of the default net peer group [line 21]:
  ```
  System.out.println ("  Group ID = " +
                      netPeerGroup.getPeerGroupID().toString());
  ```
- *Peer Name* — our peer name; whatever we entered on the JXTA Configurator basic settings [line 23]:
  ```
  System.out.println ("  Peer name = " +
                      netPeerGroup.getPeerName());
  ```

- *Peer ID* — the unique peer ID that was assigned to our JXTA peer when we ran the application [line 25]:

```
System.out.println ("  Peer ID = " +
                          netPeerGroup.getPeerID().toString());
```

After printing this information, the application calls the stopApp() method to stop the group services [line 28] and then exits.

```
myapp.netPeerGroup.stopApp();
```

Running the Hello World Example

The first time SimpleJxtaApp is run, the auto-configuration tool is displayed. After you enter the configuration information and click OK, the application continues and prints out information about the JXTA peer and peer group.

When the application completes, you can investigate the various files and subdirectories that were created in the `./.jxta` subdirectory:

- `PlatformConfig` — the configuration file created by the auto-configuration tool
- `cm` — the local cache directory; it contains subdirectories for each group that is discovered. In our example, we should see the `jxta-NetGroup` and `jxta-WorldGroup` subdirectories. These subdirectories will contain index files (`*.idx`) and advertisement store files (`advertisements.tbl`).
- `pse` — subdirectory containing your peer certificate (used for security).

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

<!DOCTYPE jxta:PA>

<jxta:PA xmlns:jxta="http://jxta.org">
    <PID>
       urn:jxta:uuid-
59616261646162614A78746150325033F3BC76FF13C2414CBC0AB663666DA53903
    </PID>
    <GID>
       urn:jxta:jxta-NetGroup
    </GID>
    <Name>
       suz
    </Name>
    <Svc>
       <MCID>
          urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000805
       </MCID>
       <Parm>
          <Addr>
             tcp://192.168.1.100:9701/
          </Addr>
          <Addr>
             jxtatls://uuid-
59616261646162614A78746150325033F3BC76FF13C2414CBC0AB663666DA53903/
TlsTransport/jxta-WorldGroup
          </Addr>
          <Addr>
             jxta://uuid-
59616261646162614A78746150325033F3BC76FF13C2414CBC0AB663666DA53903/
          </Addr>
          <Addr>
             http://JxtaHttpClientuuid-
59616261646162614A78746150325033F3BC76FF13C2414CBC0AB663666DA53903/
          </Addr>
       </Parm>
    </Svc>
</jxta:PA>
```

Figure 6-5 Example JXTA peer advertisement.

Source Code: SimpleJxtaApp

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

/**
 *  This is a simple example of how an application would start jxta
 */

public class SimpleJxtaApp {

   static PeerGroup netPeerGroup = null;

   public static void main(String args[]) {

      System.out.println ("Starting JXTA ....");
      SimpleJxtaApp myapp = new SimpleJxtaApp();
      myapp.startJxta();

      System.out.println ("Hello from JXTA group " +
                          netPeerGroup.getPeerGroupName() );
      System.out.println ("  Group ID = " +
                          netPeerGroup.getPeerGroupID().toString());
      System.out.println ("  Peer name = " +
                          netPeerGroup.getPeerName());
      System.out.println ("  Peer ID = " +
                          netPeerGroup.getPeerID().toString());
      System.out.println ("Good Bye ....");
      myapp.netPeerGroup.stopApp();
   }

   private void startJxta() {
      try {
         // create and start the default JXTA NetPeerGroup
         netPeerGroup = PeerGroupFactory.newNetPeerGroup();
      }
      catch (PeerGroupException e) {
         // could not instantiate the group, print the stack and exit
         System.out.println("fatal error : group creation failure");
         e.printStackTrace();
         System.exit(1);
      }
   }
}
```

CHAPTER **7**

Programming with JXTA

This chapter presents several JXTA programming examples that perform common tasks such as peer and peer group discovery, creating and publishing advertisements, creating and joining a peer group, and using pipes.

Note – These examples were developed and tested using the 03-02-2003 JXTA Stable Build (JXTA_2_0_Stable_20030301 03-02-2003).

Peer Discovery

This programming example illustrates how to discover other JXTA peers on the network. The application instantiates the JXTA platform, and then sends out Discovery Query messages to the default netPeerGroup looking for any JXTA peer. For each Discovery Response message received, the application prints the name of the peer sending the response (if it is known) as well as the name of each peer that was discovered.

Figure 7-1 shows example output when this application is run:

```
Sending a Discovery Message
Sending a Discovery Message
Got a Discovery Response [5 elements] from peer : unknown
 Peer name = suz
 Peer name = jsoto-2K
 Peer name = peertopeer
 Peer name = JXTA.ORG 237
 Peer name = Frog@SF05
 Sending a Discovery message
Got a Discovery Response [5 elements] from peer : unknown
 Peer name = Mr Magoo
 Peer name = mypc
 Peer name = yaro-work
 Peer name = johnboy2
 Peer name = Lomax@DIOXINE.NET
```

Figure 7-1 Example output: Peer discovery example.

Note – Because Discovery Responses are sent asynchronously, you may need to wait while several Discovery Requests are sent before receiving any responses. If you don't receive any Discovery Responses when you run this application, you most likely haven't configured your JXTA environment correctly. You will typically want to specify at least one rendezvous peer. If your peer is located behind a firewall or NAT, you will also need to specify a relay peer. Remove the `PlatformConfig` file that was created in the current directory and re-run the application. When the JXTA Configurator appears, enter the correct configuration information. See *http://platform.jxta.org/java/confighelp.html* for more details on using the JXTA Configurator tool.

Discovery Service

The JXTA DiscoveryService provides an asynchronous mechanism for discovering peer, peer group, pipe, and service advertisements. Advertisements are stored in a persistent local cache (the `./.jxta/cm` directory). When a peer boots up, the same cache is referenced. Within the `./.jxta/cm` directory, subdirectories are created for each peer group that is discovered.

- `./.jxta/cm/jxta-NetGroup` — contains advertisements for the net peer group
- `./.jxta/cm/group-ID` — contains advertisements for this group

These directories will contain files of the following types:

- `*.idx` — index files
- `record-offsets.tbl` — entry list store
- `advertisements.tbl` — advertisement store

A JXTA peer can use the getLocalAdvertisements() method to retrieve advertisements that are in its local cache. If it wants to discover other advertisements, it uses getRemoteAdvertisements() to send a Discovery Query message to other peers. Discovery Query messages can be sent to a specific peer or propagated to the JXTA network. In the J2SE platform binding, Discovery Query messages not intended for a specific peer are propagated on the local subnet utilizing IP multicast and also sent to the peer's rendezvous. Connection to the rendezvous peer occurs asynchronougly. If this peer has not yet connected to a rendezvous, the Discovery Query message will only be sent to the local subnet via multicast. Once the peer has connected to a rendezvous, the Discovery Query message will also be propagated to the rendezvous peer. A peer includes its own advertisement in the Discovery Query message, performing an announcement or automatic discovery mechanism.

There are two ways to receive DiscoveryResponse messages. You can wait for one or more peers to respond with DiscoveryResponse messages, and then make a call to getLocalAdvertisements() to retrieve any results that have been found and have been added to the local cache. Alternately, asynchronous notification of discovered peers can be accomplished by adding a Discovery Listener whose callback method, discoveryEvent(), is called when discovery events are received. If you choose to add a Discovery Listener, you have two options. You can call addDiscoveryListener() to register a listener. Or, you can pass the listener as an argument to the getRemoteAdvertisements() method.

The DiscoveryService is also used to publish advertisements. This is discussed in more detail in the example "Creating Peer Groups and Publishing Advertisements" on page 44.

The following classes are used in this example:

- *net.jxta.discovery.DiscoveryService* — asynchronous mechanism for discovering peer, peer group, pipe and service advertisements and publishing advertisements.

- *net.jxta.discovery.DiscoveryListener* — the listener interface for receiving DiscoveryService events.
- *net.jxta.DiscoveryEvent* — contains Discovery Response messages.
- *net.jxta.protocol.DiscoveryResponseMsg* — defines the Discovery Service "response"

DiscoveryDemo

This example uses the DisoveryListener interface to receive asynchronous notification of discovery events. [The code for this example begins on page 36.] We define a single class, DiscoveryDemo, which implements the DiscoveryListener interface. We also define a class variable:

- `PeerGroup netPeerGroup` — our peer group (the default net peer group) [line 14]

and four methods:

- `public void startJxta()` — initialize the JXTA platform [line 18]
- `public void run()` — thread to send DiscoveryRequest messages [line 39]
- `public void discoveryEvent(DiscoveryEvent ev)` — handle DiscoveryResponse messages that are received [line 67]
- `static public void main()` — main routine [line 96]

startJxta() method

The startJxta() method instantiates the JXTA platform (the JXTA world group) and creates the default net peer group [line 31]:

```
netPeerGroup = PeerGroupFactory.newNetPeerGroup();
```

Next, our discovery service is retrieved from our peer group, the netPeerGroup [line 31]:

```
discovery = netPeerGroup.getDiscoveryService();
```

This discovery service will be used later to add ourselves as a DiscoveryListener for DiscoveryResponse events and to send DiscoveryRequest messages.

run() method

The run() method first adds the calling object as a DiscoveryListener for DiscoveryResponse events [line 42]:

```
discovery.AddDiscoveryListener(this);
```

Now, whenever a Discovery Response message is received, the discoveryEvent() method for this object will be called. This enables our application to asynchronously be notified every time this JXTA peer receives a Discovery Response message.

Next, the run() method loops forever sending out DiscoveryRequest messages via the getRemoteAdvertisements() method. The getRemoteAdvertisements() method takes 5 arguments:

- `java.lang.string peerid` — ID of a peer to send query to; if null, propagate query request
- `int type` — DiscoveryService.PEER, DiscoveryService.GROUP, DiscoveryService.ADV
- `java.lang.string attribute` — attribute name to narrow discovery to
- `java.lang.string value` — value of attribute to narrow discovery to
- `int threshold` — the upper limit of responses from one peer

There are two main ways to send DiscoveryRequests via the Discovery Service. If a peer ID is specified in the getRemoteAdvertisement() call, the message is sent to only that one peer. In this case, the Endpoint Router attempts to resolve the destination peer's endpoints locally; if necessary, it routes the message to other relays in an attempt to reach the specified peer. If a null peer ID is specified in the getRemoteAdvertisements() call, the discovery message is propagated on the local subnet utilizing IP multicast, and the message is also propagated to the rendezvous peer. Only peers in the same peer group will respond to a DiscoveryRequest message.

The type parameter specifies which type of advertisements to look for. The DiscoveryService class defines three constants: DiscoveryService.PEER (looks for peer advertisements), DiscoveryService.GROUP (looks for peer group advertisements), and DiscoveryService.ADV (looks for all other advertisement types, such as pipe advertisements or module class advertisements).

The discovery scope can be narrowed down by specifying an Attribute and Value pair; only advertisements that match will be returned. The Attribute must exactly match an element name in the associated XML document. The Value string can use wildcards (e.g., *) to determine the match. For example, the following call would limit the search to peers whose name contained the exact string "test1":

```
discovery.getRemoteAdvertisements(null, DiscoveryService.PEER,
                                  "Name", "test1", 5);
```

while this example, using wildcards, would return any peer whose name contained the string "test":

```
discovery.getRemoteAdvertisements(null, DiscoveryService.PEER,
                                  "Name", "*test*", 5);
```

The search can also be limited by specifying a threshold value, indicating the upper limit of responses from one peer.

In our example [line 47], we send Discovery Request messages to the local subnet and the rendezvous peers, looking for any peer. By specifying a threshold value of 5, we will get a maximum of 5 responses (peer advertisements) in each Discovery Response message. If the peer has more than the specified number of matches, it will select the elements to return at random.

```
discovery.getRemoteAdvertisements(null, DiscoveryService.PEER,
                                                 null, null, 5);
```

There is no guarantee that there will be a response to a DiscoveryRequest message. A peer may receive zero, one, or more responses.

discoveryEvent() method

Because our class implements the DiscoveryListener interface, we must have a discoveryEvent() method [line 67]:

```
    public void discoveryEvent(DiscoveryEvent ev)
```

The Discovery Service calls this method whenever a DiscoveryResponse message is received. Peers that have been discovered are automatically added to the local cache (`.jxta/cm/`*`group_name`*) by the Discovery Service.

The first part of this method prints out a message reporting which peer sent the response.

The discoveryEvent method is passed a single argument of type DiscoveryEvent. The getResponse() method returns the response associated with this event. In our example, this method returns a DiscoveryResponseMsg [line 69]:

```
DiscoveryResponseMsg res = ev.getResponse();
```

Each DiscoveryResponseMsg object contains the responding peer's peer advertisement, a count of the number of responses returned, and an enumeration of peer advertisements (one for each discovered peer). Our example retrieves the responding peer's advertisement from the message [line 73]:

```
PeerAdvertisement peerAdv = res.getPeerAdvertisement();
```

Because some peers may not respond with their peer advertisement, the code checks if the peer advertisement is null. If it is not null, it extracts the responding peer's name [line 77]:

```
name = peerAdv.getName();
```

Now we print a message stating we received a response and include the name of the responding peer (or unknown, if the peer did not include its peer advertisement in its response) [line 80]:

```
System.out.println ("Got a Discovery Response [" +
                    res.getResponseCount()+ " elements]  from peer : " +
                    name);
```

The second part of this method prints out the names of each discovered peer. The responses are returned as an enumeration, and can be retrieved from the DiscoveryResponseMsg [line 86]:

```
Enumeration enum = res.getAdvertisements();
```

Each element in the enumeration is a PeerAdvertisement, and for each element we print the peer's name [line 90]:

```
adv = (PeerAdvertisement) enum.nextElement();
System.out.println (" Peer name = " + adv.getName());
```

main()

The main() method [starting at line 96] first creates a new object of class DiscoveryDemo. It then calls the startJxta() method , which instantiates the JXTA platform. Finally, it calls the run() method, which loops continuously sending out discovery requests.

Source Code: DiscoveryDemo

```
import java.util.Enumeration;

import net.jxta.discovery.DiscoveryEvent;
import net.jxta.discovery.DiscoveryListener;
import net.jxta.discovery.DiscoveryService;
import net.jxta.exception.PeerGroupException;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.protocol.DiscoveryResponseMsg;
import net.jxta.protocol.PeerAdvertisement;

public class DiscoveryDemo implements Runnable, DiscoveryListener {

   static PeerGroup netPeerGroup  = null;
   private DiscoveryService discovery;

   // start the JXTA platform
   private void startJxta() {
      try {
         netPeerGroup = PeerGroupFactory.newNetPeerGroup();
      }
      catch ( PeerGroupException e) {

         // could not instantiate the group, print the stack and exit
         System.out.println("fatal error : group creation failure");
         e.printStackTrace();
         System.exit(1);
      }

      // Get the discovery service from our peer group
      discovery = netPeerGroup.getDiscoveryService();
    }

   /**
    * This thread loops forever discovering peers
    * every minute, and displaying the results.
    */

   public void run() {
      try {
         // Add ourselves as a DiscoveryListener for DiscoveryResponse events
         discovery.addDiscoveryListener(this);

         while (true) {
            System.out.println("Sending a Discovery Message");
            // look for any peer
```

```
                discovery.getRemoteAdvertisements(null, DiscoveryService.PEER,
                        null, null, 5);

                // wait a bit before sending next discovery message
                try {
                    Thread.sleep(60 * 1000);
                } catch(Exception e) {}

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


/**
 * by implementing DiscoveryListener we must define this method
 * to deal with discovery responses
 */

    public void discoveryEvent(DiscoveryEvent ev) {

        DiscoveryResponseMsg res = ev.getResponse();
        String name = "unknown";

        // Get the responding peer's advertisement
        PeerAdvertisement peerAdv = res.getPeerAdvertisement();

        // some peers may not respond with their peerAdv
        if (peerAdv != null) {
            name = peerAdv.getName();
        }

        System.out.println ("Got a Discovery Response [" +
                        res.getResponseCount()+ " elements]  from peer : " +
                        name);

        // printout each discovered peer
        PeerAdvertisement adv = null;
        Enumeration enum = res.getAdvertisements();

        if (enum != null ) {
           while (enum.hasMoreElements()) {
              adv = (PeerAdvertisement) enum.nextElement();
              System.out.println (" Peer name = " + adv.getName());
           }
        }
     }

     static public void main(String args[]) {
        DiscoveryDemo myapp  = new DiscoveryDemo();
        myapp.startJxta();
        myapp.run();
     }
  }
```

Peer Group Discovery

Peer group discovery is very similar to the peer discovery in the previous example. The primary difference is that instead of sending DiscoveryRequest messages looking for peers, we send DiscoveryRequest messages looking for peer groups. Any DiscoveryResponse messages we receive will contain peer group advertisements rather than peer advertisements.

In this example, however, after instantiating the JXTA platform we wait until we are connected to a rendezvous peer before sending Discovery Request messages. It would not be necessary to wait for a rendezvous connection if your application was running locally on a subnet and communicating with other peers on that subnet via multicast. However, in other configurations you might want to wait until a rendezvous connection is established before sending requests.

For each DiscoveryResponse message received, the application prints the name of the peer sending the response (if it is known) as well as the name of each peer group that was discovered.

Figure 7-2 shows example output when this application is run:

```
Waiting to connect to rendezvous...connected!
Sending a Discovery message
Sending a Discovery message
 Got a Discovery Response [6 elements] from peer : unknown
 Peer Group = football
 Peer Group = weaving
 Peer Group = P2P-discuss
 Peer Group = genome
 Peer Group = mygroup
 Peer Group = baseball
Sending a Discovery message
 Got a Discovery Response [4 elements] from peer : unknown
 Peer Group = testgroup1
 Peer Group = soccer
 Peer Group = osa_test
 Peer Group = travel
```

Figure 7-2 Example output: Peer group discovery example.

Source code for this example begins on page 41. Differences from the previous peer discovery example are indicated in bold font.

startJxta() method

The first part of this method is identical to that of the previous Peer Discovery example — it instantiates the net peer group and extracts the discovery service from the peer group. However, this example also extracts the RendezVousService from the peer group:

```
rdv = netPeerGroup.getRendezVousService();
```

Then it loops, waiting until a connection is established to a rendezvous peer. The method isConnectedToRendezVous() returns true if this peer is currently connected to a rendezvous; otherwise, it returns false. [line 42]:

```
while (! rdv.isConnectedToRendezVous()) {
```

run() method

The only difference in this method is that we send out DiscoveryRequest messages looking for peer groups, rather than peers [line 65]:

```
        discovery.getRemoteAdvertisements(null, DiscoveryService.GROUP,
                                    null, null, 5;
```

The remainder of the code is identical to the peer discovery example.

discoveryEvent() method

The first part of this method is identical to the peer discovery example: we retrieve the DiscoveryResponseMsg, extract the responding peer's advertisement, and then print a message stating the name of the responding peer (if it is known) and the number of responses received.

The changes occur in the second part of the method, which prints out the names of each discovered peer group. Like the peer discovery example, responses are returned as an enumeration and are retrieved from the DiscoveryResponseMsg [line 101]:

```
        Enumeration enum = res.getAdvertisements();
```

Now, instead of receiving an enumeration of peer advertisements, we receive an enumeration peer group advertisements [line 105]:

```
        adv = (PeerGroupAdvertisement) enum.nextElement();
        System.out.println (" Peer Group = " + adv.getName());
```

Source Code: GroupDiscoveryDemo

```java
import java.util.Enumeration;

import net.jxta.discovery.DiscoveryEvent;
import net.jxta.discovery.DiscoveryListener;
import net.jxta.discovery.DiscoveryService;
import net.jxta.exception.PeerGroupException;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.protocol.DiscoveryResponseMsg;
import net.jxta.protocol.PeerAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;
import net.jxta.rendezvous.RendezVousService;

public class GroupDiscoveryDemo implements DiscoveryListener {

    static PeerGroup netPeerGroup  = null;
    private DiscoveryService discovery;
    private RendezVousService rdv;

   /**
    *  Method to start the JXTA platform.
    *  Waits until a connection to rdv is established.
    */
   private void startJxta() {
      try {
         netPeerGroup = PeerGroupFactory.newNetPeerGroup();
      }
      catch ( PeerGroupException e) {

         // could not instantiate the group, print the stack and exit
         System.out.println("fatal error : group creation failure");
         e.printStackTrace();
         System.exit(1);
      }

      // Extract the discovery and rendezvous services from our peer group
      discovery = netPeerGroup.getDiscoveryService();
      rdv = netPeerGroup.getRendezVousService();

      // Wait until we connect to a rendezvous peer
      System.out.print("Waiting to connect to rendezvous...");
      while (! rdv.isConnectedToRendezVous()) {
         try {
            Thread.sleep(2000);
         } catch (InterruptedException ex) {
            // nothing, keep going
         }
```

```
        }
        System.out.println("connected!");
    }

    /**
     * This thread loops forever discovering peers
     * every minute, and displaying the results.
     */

    public void run() {

        try {
            // Add ourselves as a DiscoveryListener for DiscoveryResponse events
            discovery.addDiscoveryListener(this);
            while (true) {
                System.out.println("Sending a Discovery Message");
                // look for any peer group
                discovery.getRemoteAdvertisements(null, DiscoveryService.GROUP,
                        null, null, 5);

                // wait a bit before sending next discovery message
                try {
                    Thread.sleep(60 * 1000);
                } catch(Exception e) {}
            }
        }
        catch(Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * by implementing DiscoveryListener we must define this method
     * to deal to discovery responses
     */

    public void discoveryEvent(DiscoveryEvent ev) {

        DiscoveryResponseMsg res = ev.getResponse();
        String name = "unknown";

        // Get the responding peer's advertisement
        PeerAdvertisement peerAdv = res.getPeerAdvertisement();
        // some peers may not respond with their peerAdv
        if (peerAdv != null) {
            name = peerAdv.getName();
        }
        System.out.println (" Got a Discovery Response [" +
                res.getResponseCount()+ " elements]  from peer : " +
                        name);

        // now print out each discovered peer group
        PeerGroupAdvertisement adv = null;
        Enumeration enum = res.getAdvertisements();

        if (enum != null ) {
           while (enum.hasMoreElements()) {
              adv = (PeerGroupAdvertisement) enum.nextElement();
              System.out.println (" Peer Group = " + adv.getName());
           }
        }
    }

    static public void main(String args[]) {
        GroupDiscoveryDemo myapp  = new GroupDiscoveryDemo();
        myapp.startJxta();
        myapp.run();
    }

}
```

Creating Peer Groups and Publishing Advertisements

This example first prints the names and IDs of all peer groups in the local cache. The first time this application is run, there should be no peer groups in the local cache. Then, it creates a new peer group, prints its group name and group ID, and publishes its advertisement. Finally, it prints the names and IDs of all peer groups now in the local cache.

Figure 7-3 shows example output when this application is run:

```
--- local cache (Peer Groups)  ---
--- end local cache ---
Creating a new group advertisement
  Group = PubTest
  Group ID = urn:jxta:uuid-791A0C3A50CE43D891E0BDC5689CC902
Group published successfully.
--- local cache (Peer Groups)  ---
PubTest, group ID = urn:jxta:uuid-791A0C3A50CE43D891E0BDC5689CC902
--- end local cache ---
```

Figure 7-3 Example output: Peer group creation/publishing example.

The peer group advertisement that we create is added to the local cache directory, `.jxta/cm`. In addition, a new directory with the same name as the peer group ID is created, and this directory contains advertisements that are discovered in the context of this new peer group. An advertisement for our peer is added to this cache directory. Advertisements for any additional peers that are discovered in the new peer group would also be added here.

- `./.jxta/cm/jxta-NetGroup` — local cache directory containing advertisements for the net peer group
- `./.jxta/cm/1D5E451AF1B243C1AD39B9D331AE858C02` — cache directory for the new peer group

main()

This method calls startJxta() to instantiate the JXTA platform and create the default netPeerGroup. It then calls groupsInLocalCache() to display the names and IDs of all groups currently in the local cache (this should be empty the first time this application is run). Next, it calls createGroup() to create a new JXTA peer group and to publish the new peer group's advertisement. Finally, it calls groupsInLocalCache() again to display the names and IDS of all groups now in the local cache. The group that we just created and published should be displayed.

startJxta()

This method is identical to earlier examples. It instantiates the JXTA platform and extracts information needed later in the application:

- Instantiates the JXTA platform and creates the default net peer group [line 41]:
  ```
  myGroup = PeerGroupFactory.newNetPeerGroup();
  ```
- Extracts the discovery service from the peer group; this is used later to publish the new group advertisement [line 50]:
  ```
  discoSvc = myGroup.getDiscoveryService();
  ```

groupsInLocalCache()

This method prints the names and IDs of all groups in the local cache. It first calls the getLocalAdvertisements() method to retrieve advertisements in the local cache. The getLocalAdvertisements() method takes 3 arguments:

- `int type` — DiscoveryService.PEER, DiscoveryService.GROUP, DiscoveryService.ADV
- `java.lang.string attribute` — attribute name to narrow discovery to
- `java.lang.string value` — value of attribute to narrow discovery to

In our example. we are looking for all peer group advertisements in the local cache [line 48]:

```
Enumeration enum = discoSvc.getLocalAdvertisements(discoSvc.GROUP,
                                                   null, null);
```

This method returns an enumeration of peer group advertisements. We step through the enumeration, printing out the name and peer group ID of each element [line 52]:

```
adv = (PeerGroupAdvertisement) enum.nextElement();
System.out.println( adv.getName() + ", group ID = " +
                   adv.getPeerGroupID().toString());
```

createGroup()

This method is used to create a new peer group and publish its advertisement.

The first part of this method [lines 68 to 76] creates the new peer group. First, we call getAllPurposePeerGroupImpleAdvertisement() to create a ModuleImplAdvertisenent, which contains entries for all of the core peer group services:

```
ModuleImplAdvertisement implAdv =
        myGroup.getAllPurposePeerGroupImplAdvertisement();
```

Next, we use newGroup() to create a new peergroup:

```
PeerGroup pg = myGroup.newGroup(null,                 // Assign new group ID
                                implAdv,              // The implem. adv
                                "PubTest",            // The name
                                "testing group adv"); // Helpful descr.
```

We pass four arguments to newGroup():

- `PeerGroup ID gid` — the peer group ID of the group to be created; if null, a new peer group ID is generated
- `Advertisement implAdv` — the implementation advertisement
- `String name` — the name of the new group
- `String description` — a group description

When a new group is created with DiscoveryService.newGroup(), its advertisement is always added to the local cache (i.e., it is published locally). It uses the default values for advertisement expiration: a local lifetime (the time the advertisement is going to be kept locally on the peer that originally created it) of 365 days, and a remote lifetime (the time the advertisement is going to be kept in the cache of peers that have searched and retrieved the advertisement) of two hours.

Note – Since the DiscoveryService.newGroup() method publishes the new group for us, it is not necessary to explicitly call DiscoveryService.publish().

After the group is created, we print the name of the group and its peer group ID.

The second part of this method publishes the new peer group advertisement remotely [line 91]:

```
    discoSvc.remotePublish(adv, DiscoveryService.GROUP);
```

This method takes two arguments: the advertisement to be published and the advertisement type. It uses the default advertisement expiration. This call uses the discovery service to send messages on the local subnet and also to the rendezvous peer.

Note – If the peer is not connected to a rendezvous when the remotePublish() method is called, the peer group advertisement will be sent only to peers on the local subnet via multicast. If the peer is connected to a rendezvous when the remotePublish() method is called, the peer group advertisement will also be sent to the rendezvous peer. If it is important to publish the group advertisement outside the local subnet, you should ensure that you are connected to a rendezvous peer before calling the remotePublish() method. (For more information on waiting until a connection to a rendezvous is established, please see "Peer Group Discovery" on page 39.)

Source Code: PublishDemo

```
import java.util.Enumeration;

import net.jxta.discovery.DiscoveryService;
import net.jxta.exception.PeerGroupException;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.protocol.PeerGroupAdvertisement;
import net.jxta.protocol.ModuleImplAdvertisement;

public class PublishDemo   {

    static PeerGroup myGroup = null;
    private DiscoveryService discoSvc;

    public static void main(String args[]) {
       PublishDemo myapp = new PublishDemo();
       System.out.println ("Starting PublishDemo ....");
       myapp.startJxta();
       myapp.groupsInLocalCache();
       myapp.createGroup();
       myapp.groupsInLocalCache();
       System.exit(0);
    }

   private void startJxta() {
       try {
           // create, and start the default jxta NetPeerGroup
           myGroup = PeerGroupFactory.newNetPeerGroup();
       } catch (PeerGroupException e) {
           // could not instantiate the group, print the stack and exit
           System.out.println("fatal error : group creation failure");
           e.printStackTrace();
           System.exit(1);
       }

       // obtain the the discovery service
       discoSvc = myGroup.getDiscoveryService();
   }

```

```
     // print all peer groups found in the local cache
    private void groupsInLocalCache() {

       System.out.println("--- local cache (Peer Groups)  ---");

       try {
          PeerGroupAdvertisement adv = null;
          Enumeration enum = discoSvc.getLocalAdvertisements(discoSvc.GROUP,
                   null, null);
          if (enum != null) {
             while (enum.hasMoreElements()) {
                adv = (PeerGroupAdvertisement) enum.nextElement();
             System.out.println( adv.getName() + ", group ID = " +
                   adv.getPeerGroupID().toString());
             }
          }
       } catch (Exception e) {}

       System.out.println("--- end local cache ---");
    }

    // create and publish a new peer group
    private void createGroup() {
       PeerGroupAdvertisement adv;

       System.out.println("Creating a new group advertisement");

       try {
          // create a new all purpose peergroup.
          ModuleImplAdvertisement implAdv =
                   myGroup.getAllPurposePeerGroupImplAdvertisement();

          PeerGroup pg = myGroup.newGroup(null,         // Assign new group ID
                                         implAdv,          // The implem. adv
                                          "PubTest",         // The name
                                        "testing group adv"); // Helpful descr.

          // print the name of the group and the peer group ID
          adv = pg.getPeerGroupAdvertisement();
          PeerGroupID GID = adv.getPeerGroupID();
          System.out.println("  Group = " +adv.getName() +
                            "\n  Group ID = " + GID.toString());
       } catch (Exception eee) {
          System.out.println("Group creation failed with " + eee.toString());
          return;
       }

```

```
        try {
            // publish this advertisement
            //(send out to other peers and rendezvous peer)
            discoSvc.remotePublish(adv, DiscoveryService.GROUP);
            System.out.println("Group published successfully.");
        }
        catch (Exception e) {
            System.out.println("Error publishing group advertisement");
            e.printStackTrace();
            return;
        }
    }
}
```

Joining a Peer Group

This example creates and publishes a new peer group, joins the peer group, and prints its authorization credential.

Figure 7-4 shows example output when this application is run:

```
Starting JoinDemo ....
Creating a new group advertisement
  Group = JoinTest
  Group ID = urn:jxta:uuid-1D5E451AF1B243C1AD39B9D331AE858C02
Group published successfully.

Joining peer group...
Successfully joined group JoinTest

Credential:
NullCredential :
       PeerGroupID: urn:jxta:uuid-1D5E451AF1B243C1AD39B9D331AE858C02
       PeerID : urn:jxta:uuid-59616261646162614A78746150325033F3B
C76FF13C2414CBC0AB663666DA53903
       Identity : nobody

Good Bye ....
```

Figure 7-4 Example output: Creating and joining a peer group.

This example builds upon the previous example which created and published a new group. The new code in this example is in the joinGroup() method, which illustrates how to apply for group membership and then join a group. This example uses the default mechanism for joining a group. An example of how to join a secure group is included later in this document (see “Creating a Secure Peer Group” on page 102).

Membership Service

In JXTA, the Membership Service is used to apply for peer group membership, join a peer group, and resign from a peer group. The membership service allows a peer to establish an identity within a peer group. Once an identity has been established, a credential is available which allows the peer to prove that it rightfully has that identity. Identities are used by services to determine the capabilities which should be offered to peers.

When a peer group is instantiated on a peer, the membership service for that peer group establishes a default temporary identity for the peer within the peergroup. This identity, by convention, only allows the peer to establish its true identity.

The sequence for establishing an identity for a peer within a peer group is as follows:

- *Apply*

  The peer provides the membership service an initial credential which may be used by the service to determine which method of authentication is to be used to establish the identity of this peer. If the service allows authentication using the requested mechanism, then an appropriate authenticator object is returned.

The peer group instance is assumed to know how to interact with the authenticator object (remember that it requested the authentication method earlier in the apply process).

- *Join*

  The completed authenticator is returned to the Membership Service and the identity of this peer is adjusted based on the new credential available from the authenticator. The identity of the peer remains as it was until the Join operation completes.

- *Resign*

  Whatever existing identity that is established for this peer is discarded and the current identity reverts to the "nobody" identity.

Authentication credentials are used by the JXTA MembershipService services as the basis for applications for peer group membership. The AuthenticationCredential provides two important pieces of information: the authentication method being requested and the identity information which will be provided to that authentication method. Not all authentication methods use the identity information.

main()

This method calls the remaining three class methods:

- startJxta() — to instantiate the JXTA platform and create the default net peer group [line 29]
- createGroup() — to create and publish a new peer group [line 30]
- joinGroup() — to join the new group [line 32]

startJxta()

This method [line 38] is identical to the startJxta() method in previous examples: it instantiates the JXTA platform and creates the default netPeerGroup, and extracts our discovery service from the netPeerGroup. The discovery service will be used later to publish the peer group we create.

createGroup()

This method [line 53] is almost identical to the createGroup() method in the previous example (see description on page 45). It is used to create a new peer group and publish its advertisement. The only significant change is that if the group is successfully created, this method returns the new PeerGroup. If there is an error creating the new peer group, this method returns null.

joinGroup()

This method is used to join the peer group that is passed as an argument [line 97]:

```
private void joinGroup(PeerGroup grp)
```

In the example code, the joinGroup() method first generates the authentication credentials for the peer in the specified peer group [line 104]:

```
AuthenticationCredential authCred =
   new AuthenticationCredential( grp, null, creds );
```

This constructor takes three arguments:

- `PeerGroup peergroup` — the peer group context in which this AuthenticationCredential is created (i.e., the peer group that you want to join).

- `java.lang.String method` — The authentication method which will be requested when the AuthenticationCredential is provided to the peer group MembershipService service.
- `Element IdentityInfo` — Optional additional information about the identity being requested, which is used by the authentication method. This information is passed to the authentication method during the apply operation of the MembershipService service.

AuthenticationCredentials are created in the context of a PeerGroup. However, they are generally independent of peer groups. The intent is that the AuthenticationCredential will be passed to the MembershipService of the same peer group.

Next, our example extracts the MembershipService from the peer group we want to join [line 108]:

```
MembershipService membership = grp.getMembershipService();
```

And uses the MembershipService.apply() method to apply for group membership [line 111]:

```
Authenticator auth = membership.apply( authCred );
```

The authentication credentials created earlier in the method are passed to the apply() method. Included in the credentials is information about our peer group ID, our peer ID, and our identity to be used when joining this group. The apply method returns an Authenticator object, which is used to check if authentication has completed correctly. The mechanism for completing the authentication object is unique for each authentication method. The only common operation is isReadyForJoin(), which provides information on whether the authentication process has completed correctly.

After applying for membership, the next step is to join the group. First, the Authenticator.isReadyForJoin() method is called to verify the authentication process. This method returns true if the authenticator object is complete and ready for submitting to the MembershipService service for joining; otherwise, it returns false. If everything is okay to join the group, the MembershipService.join() method is called to join the group [line 114]:

```
if (auth.isReadyForJoin()){
   Credential myCred = membership.join(auth);
```

The MembershipService.join() method returns a Credential object.

Note – Some authenticators may behave asynchronously, and this method can be used to determine if the authentication process has completed. This method makes no distinction between incomplete authentication and failed authentication.

Note – When a peer joins a peer group, it will automatically seek a rendezvous peer for that peer group. If it finds no rendezvous peer, it will dynamically become a rendezvous for this peer group.

Source Code: JoinDemo

```java
import java.io.StringWriter;

import net.jxta.credential.AuthenticationCredential;
import net.jxta.credential.Credential;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.MimeMediaType;
import net.jxta.membership.Authenticator;
import net.jxta.membership.MembershipService;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.protocol.PeerGroupAdvertisement;
import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.discovery.DiscoveryService;
import net.jxta.exception.PeerGroupException;

public class JoinDemo   {

    static PeerGroup myGroup = null;    // my initial group
    private DiscoveryService discoSvc;


    public static void main(String args[]) {

        System.out.println ("Starting JoinDemo ....");
        JoinDemo myapp = new JoinDemo();

        myapp.startJxta();
        PeerGroup newGroup = myapp.createGroup();
        if (newGroup != null) {
            myapp.joinGroup(newGroup);
        }
        System.out.println ("Good Bye ....");
        System.exit(0);
    }

    private void startJxta() {
        try {
            // create, and Start the default jxta NetPeerGroup
            myGroup = PeerGroupFactory.newNetPeerGroup();
        } catch (PeerGroupException e) {
            // could not instantiate the group, print the stack and exit
            System.out.println("fatal error : group creation failure");
            e.printStackTrace();
            System.exit(1);
        }

        // Extract the discovery service from our peer group
        discoSvc = myGroup.getDiscoveryService();
      }

    private PeerGroup createGroup() {
        PeerGroup pg;               // new peer group
        PeerGroupAdvertisement adv; // advertisement for the new peer group

   System.out.println("Creating a new group advertisement");

        try {
          // create a new all purpose peergroup.
          ModuleImplAdvertisement implAdv =
                    myGroup.getAllPurposePeerGroupImplAdvertisement();

          pg = myGroup.newGroup(null,                // Assign new group ID
                                      implAdv,             // The implem. adv
                                         "JoinTest",            // The name
                                      "testing group adv"); // Helpful descr.

          // print the name of the group and the peer group ID
          adv = pg.getPeerGroupAdvertisement();
          PeerGroupID GID = adv.getPeerGroupID();
          System.out.println("  Group = " +adv.getName() +
                             "\n  Group ID = " + GID.toString());

        }
        catch (Exception eee) {
           System.out.println("Group creation failed with " + eee.toString());
            return (null);
       }

        try {
            // publish this advertisement
            // (send out to other peers/rendezvous peers)
            discoSvc.remotePublish(adv, DiscoveryService.GROUP);
            System.out.println("Group published successfully.\n");
        }
        catch (Exception e) {
           System.out.println("Error publishing group advertisement");
           e.printStackTrace();
           return (null);
        }

        return(pg);

    }

    private void joinGroup(PeerGroup grp) {
        System.out.println("Joining peer group...");

        StructuredDocument creds = null;

        try {
            // Generate the credentials for the Peer Group
            AuthenticationCredential authCred =
                     new AuthenticationCredential( grp, null, creds );

            // Get the MembershipService from the peer group
            MembershipService membership = grp.getMembershipService();

            // Get the Authenticator from the Authentication creds
            Authenticator auth = membership.apply( authCred );

            // Check if everything is okay to join the group
            if (auth.isReadyForJoin()){
                Credential myCred = membership.join(auth);

                System.out.println("Successfully joined group " +
                     grp.getPeerGroupName());

                // display the credential as a plain text document.
                System.out.println("\nCredential: ");
                StructuredTextDocument doc = (StructuredTextDocument)
                    myCred.getDocument(new MimeMediaType("text/plain"));

                StringWriter out = new StringWriter();
                doc.sendToWriter(out);
                System.out.println(out.toString());
                out.close();

            }
            else
                System.out.println("Failure: unable to join group");

          }
          catch (Exception e){
            System.out.println("Failure in authentication.");
            e.printStackTrace();
          }
     }
}
```

Sending Messages Between Two Peers

Note – This example requires a patch to the 03-02-2003 JXTA Stable Build in order to run correctly (see issue #591, http://platform.jxta.org/issues/show_bug.cgi?id=591). The patch is included with the companion tutorial examples.

This example illustrates how to use pipes to send messages between two JXTA peers, and also shows how to implement the RendezvousListener interface. Two separate applications are used in this example:

- PipeListener — Reads in a pipe advertisement from a file (`examplepipe.adv`), creates an input pipe, and listens for messages on this pipe
- PipeExample — Reads in a pipe advertisement from a file (`examplepipe.adv`), creates an output pipe, and sends a message on this pipe

Figure 7-5 shows example output when the PipeListener application is run, and Figure 7-6 shows example output from the PipeExample application:

```
Reading in examplepipe.adv
Creating input pipe
Waiting for msgs on input pipe
Received message: Hello from peer suz-pipe[Wed Mar 26 16:27:15 PST 2003]
   message received at: Wed Mar 26 16:27:16 PST 2003
```

Figure 7-5 Example output: PipeListener.

```
Reading in examplepipe.adv
Attempting to create an OutputPipe...
Waiting for Rendezvous Connection
Got an output pipe event
    Sending message: Hello from peer suz-pipe[Wed Mar 26 16:27:15 PST 2003]
```

Figure 7-6 Example output: PipeExample.

Note – If you are running both applications on the same system, you will need to run each application from a separate subdirectory so that they can be configured to use separate ports.

The following section provides background information on the JXTA pipe service, input pipes, and output pipes. The PipeListener example begins on page 58; PipeExample begins on page 64.

JXTA Pipe Service

The PipeService class defines a set of interfaces to create and access pipes within a peer group. Pipes are the core mechanism for exchanging messages between two JXTA applications or services. Pipes provide a simple, uni-directional and asynchronous channel of communication between two peers. JXTA messages are exchanged between input pipes and output pipes. An application that wants to open a receiving communication with other peers creates an input pipe and binds it to a specific pipe advertisement. The application then publishes the pipe advertisement so that other applications or services can obtain the advertisement and create corresponding output pipes to send messages to that input pipe.

Pipes are uniquely identified throughout the JXTA world by a PipeId (UUID) enclosed in a pipe advertisement. This unique PipeID is used to create the association between input and output pipes.

Pipes are non-localized communication channels that are not bound to specific peers. This is a unique feature of JXTA pipes. The mechanism to resolve the location of pipes to a physical peer is done in a completely decentralized manner in JXTA via the JXTA Pipe Binding Protocol. The Pipe Binding Protocol does not rely on a centralized protocol such as DNS (bind Hostname to IP) to bind a pipe advertisement (i.e., symbolic name) to an instance of a pipe on a physical peer (i.e., IP address). Instead, the resolver protocol uses a dynamic and adaptive search mechanism that attempts at all times to find the peers where an instance of that pipe is running.

The following classes are used in the PipeListener and PipeExample applications:

- *net.jxta.pipe.PipeService* — defines the API to the JXTA Pipe Service.
- *net.jxta.pipe.InputPipe* — defines the interface for receiving messages from a PipeService. An application that wants to receive messages from a pipe will create an input pipe. An InputPipe is created and returned by the PipeService.
- *net.jxta.pipe.PipeMsgListener* — the listener interface for receiving PipeMsgEvent events.
- *net.jxta.pipe.PipeMsgEvent* — contains events received on a pipe.
- *net.jxta.pipe.OutputPipe* — defines the interface for sending messages from a PipeService. Applications that want to send messages onto a Pipe must first get an OutputPipe from the PipeService.
- *net.jxta.pipe.OutputPipeListener* —the listener interface for receiving OutputPipe resolution events.
- *net.jxta.pipe.OutputPipeEvent* — contains events received when an output pipe is resolved.
- *net.jxta.endpoint.Message* — defines the interface of messages sent or received to and from pipes using the PipeService API. A message contains a set MessageElements. Each MessageElement contains a namespace, name, data, and signature.

PipeListener

This application creates and listens for messages on an input pipe. It defines a single class, PipeListener, which implements the PipeMsgListener interface. Two class constants contain information about the pipe to be created:

- `String FILENAME` — the XML file containing the text representation of our pipe advertisement. (This file must exist, and must contain a valid pipe advertisement, in order for our application to run correctly.)
- `String TAG` — the message element name, or tag, which we are expecting in any message we receive

We also define four instance fields:

- `PeerGroup netPeerGroup` — our peer group, the default net peer group
- `PipeService pipeSvc` — the pipe service we use to create the input pipe and listen for messages
- `PipeAdvertisement pipeAdv` — the pipe advertisement we use to create our input pipe
- `InputPipe pipeIn` — the input pipe that we create

main()

This method [line 30] creates a new PipeListener object, calls startJxta() to instantiate the JXTA platform and create the default net peer group, and then calls run() which creates the input pipe and registers this object as a PipeMsgListener. (Note: This application never ends, because of the "invisible" Java thread which does the input pipe event dispatching.)

startJxta()

This method instantiates the JXTA platform and creates the default net peer group [line 42]:

```
netPeerGroup = PeerGroupFactory.newNetPeerGroup();
```

Then it retrieves the PipeService from the default net peer group [line 51]. This service is used later when we create an input pipe:

```
pipeSvc = netPeerGroup.getPipeService();
```

Next, we create a pipe advertisement by reading it in from the existing file `examplepipe.adv` [line 54]:

```
FileInputStream is = new FileInputStream(FILENAME);
```

The file `examplepipe.adv` must exist and it must be valid XML document containing a pipe advertisement, or an exception is raised by the JXTA platform. Both this application (which creates the input pipe) and the partner application (which creates the output pipe) read their pipe advertisement from the same file. The contents of the `examplepipe.adv` file are listed in Figure 7-7 on page 71.

The AdvertisementFactory.newAdvertisement() method is called to create a new pipe advertisement [line 55]:

```
pipeAdv = (PipeAdvertisement)
    AdvertisementFactory.newAdvertisement(new MimeMediaType("text/xml"),
                                                                    is);
```

The two arguments to AdvertisementFactory.newAdvertisement() are the MIME type ("text/xml" in this example) to associate with the resulting StructuredDocument (i.e. advertisement) and the InputStream containing the body of the advertisement. The type of the advertisement is determined by reading the input stream.

After the pipe advertisement is created, the input stream is closed [line 58] and the method returns:

```
is.close();
```

run()

This method uses the PipeService.createInputPipe() to create a new input pipe for our application [line 74]:

```
pipeIn = pipeSvc.createInputPipe(pipeAdv, this);
```

Because we want to listen for input pipe events, we call createInputPipe() with two arguments:

- `PipeAdvertisement adv` — the advertisement of the pipe to be created
- `PipeMsgListener listener` — the object which will receive input pipe event messages

By registering our object as a listener when we create the input pipe, our method pipeMsgEvent() will be called asynchronously whenever a pipeMsgEvent occurs on this pipe (i.e., whenever a message is received).

pipeMsgEvent()

This method [line 89] is called asynchronously whenever a pipe event occurs on our input pipe. This method is passed one argument:

- `PipeMsgEvent event` — the event that occurred on the pipe

Our method first calls PipeMsgEvent.getMessage() to retrieve the message associated with the event [line 94]:

```
msg = event.getMessage();
```

Each message contains zero or more elements, each with an associated element name (or tag) and corresponding data string. Our method calls Message.getMessageElement() to extract the elment with the specified namespace and name [line 103]:

```
        MessageElement el = msg.getMessageElement(null, TAG);
```

If an element with the specified namespace/name is not present within the message, this method returns null.

Recall that both the input pipe and the output pipe must agree on the namespace and the element name, or tag, that is used in the messages. In our example, we use the default (null) namespace and we set a constant in the PipeListener class to refer to the message element name [line 23]:

```
        private final static String TAG = "PipeListenerMsg";
```

Finally, our method prints out a message with the current time and the message that was received [line 108]:

```
       System.out.println("Received message: " + el.toString());
       System.out.println("   message received at: " + date.toString());
```

Source Code: PipeListener

```java
import java.io.FileInputStream;
import java.util.Date;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.MessageElement;
import net.jxta.exception.PeerGroupException;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.PipeMsgEvent;
import net.jxta.pipe.PipeMsgListener;
import net.jxta.pipe.PipeService;
import net.jxta.protocol.PipeAdvertisement;

// This application creates an instance of an input pipe,
// and waits for messages on the pipe

public class PipeListener implements PipeMsgListener {

    static PeerGroup netPeerGroup = null;
    private final static String TAG = "PipeListenerMsg";
    private final static String FILENAME = "examplepipe.adv";

     private PipeService pipeSvc;
     private PipeAdvertisement pipeAdv;
     private InputPipe pipeIn = null;

     public static void main(String args[]) {

         PipeListener myapp = new PipeListener();
         myapp.startJxta();
         myapp.run();
     }

    // Starts jxta

    private void startJxta() {
        try {
            // create, and Start the default jxta NetPeerGroup
            netPeerGroup = PeerGroupFactory.newNetPeerGroup();
        }
        catch (PeerGroupException e) {
            // could not instantiate the group, print the stack and exit
            System.out.println("Error: fatal error : group creation failure");
            e.printStackTrace();
```

```
          System.exit(1);
      }

      pipeSvc = netPeerGroup.getPipeService();
      System.out.println("Reading in " + FILENAME);
      try {
         FileInputStream is = new FileInputStream(FILENAME);
         pipeAdv = (PipeAdvertisement)
            AdvertisementFactory.newAdvertisement(
            new MimeMediaType("text/xml"), is);
         is.close();
      }
      catch (Exception e) {
         System.out.println("Error: failed to read/parse pipe adv.");
         e.printStackTrace();
         System.exit(-1);
      }
   }

   // create input pipe, and register as a PipeMsgListener to be
   // asynchronously notified of any messaged received on this input pipe

   public void run() {

      try {
         System.out.println("Creating input pipe");
         pipeIn = pipeSvc.createInputPipe(pipeAdv, this);
      } catch (Exception e) {
         System.out.println("Error creating input pipe.");
         return;
      }
      if (pipeIn == null) {
         System.out.println("Error: cannot open InputPipe");
         System.exit(-1);
      }
      System.out.println("Waiting for msgs on input pipe");
   }

   // By implementing PipeMsgListener, we define this method to deal with
   // messages as they occur

   public void pipeMsgEvent ( PipeMsgEvent event ){

      Message msg=null;
      try {
         // grab the message from the event
         msg = event.getMessage();
         if (msg == null)
            return;
```

```
        } catch (Exception e) {
            e.printStackTrace();
            return;
        }

        // Get message element with our TAG/name
        MessageElement el = msg.getMessageElement(null, TAG);
        if (el.toString() == null)
            System.out.println("null msg received.");
        else {
            Date date = new Date(System.currentTimeMillis());
            System.out.println("Received message: " + el.toString());
            System.out.println("   message received at: " + date.toString());
        }
    }
}
```

PipeExample

This example creates an output pipe and sends a message on it. It defines a single class, PipeExample, which implements the Runnable, OutputPipeListener, and RendezvousListener interfaces. Like the partner class, PipeListener, it defines two class constants to contain information about the pipe to be created:

- `String FILENAME` — the XML file containing the text representation of our pipe advertisement
- `String TAG` — the message element name, or tag, which we will include in any message that we send

main()

This method [line 38] creates a new PipeExample object, calls startJxta() to instantiate the JXTA platform and create the default net peer group, and then calls run() which creates the output pipe.

run()

This method uses the PipeService.createOutputPipe() to create a new output pipe with a listener for our application [line 52]:

```
pipeSvc.createOutputPipe(pipeAdv, this);
```

Because we want to be notified when the pipe endpoints are resolved, we call createOutputPipe() with two arguments:

- `PipeAdvertisement adv` — the advertisement of the pipe to be created
- `OutputPipeListener listener` — the listener to be called back when the pipe is resolved

By registering our object as a listener when we create the output pipe, our method outputPipeEvent() will be called asynchronously when the pipe endpoints are resolved.

We then check if we are connected to a JXTA rendezvous peer [line 56]:

```
if ( !rdvSvc.isConnectedToRendezVous() ) {
```

If we are not connected, we call wait() to wait until we receive notification that we have connected to a rendezvous peer. Then, we send a second request to create an OutputPipe.

outputPipeEvent()

Because we implemented the OutputPipeListener interface, we must define the outputPipeEvent() method. This method [line 77] is called asynchronously by the JXTA platform when our pipe endpoints are resolved. This method is passed one argument:

- `OutputPipeEvent event` — the event that occurred on this pipe

Our method first calls OutputPipeEvent.getOutputPipe() to retrieve the output pipe that was created [line 80]:

```
OutputPipe op = event.getOutputPipe();
```

Next, we begin to assemble the message we want to send. We create the String, containing our peer name and the current time, to send. Then, we create an empty Message [line 90]:

```
msg = new Message();
```

Each message contains zero or more elements, each with an associated element namespace, name (or tag), and corresponding string. Both the input pipe and the output pipe must agree on the element namespace and name that are used in the messages. In this example, we will use the default (null) namespace. Recall that we set a constant in both the PipeListener class and the PipeExample class to contain the element name:

```
private final static String TAG = "PipeListenerMsg";
```

We next create a new StringMessageElement. The constructor takes three argument: the element tag (or name), the data, and a signature [line 91]:

```
StringMessageElement sme = new StringMessageElement(TAG, myMsg, null);
```

After creating our new MessageElement, we add it our our Message. In this example, we add our element to the null namespace [line 92]:

```
msg.addMessageElement(null, sme);
```

Now that our message object is created and it contains our text message, we send it on the output pipe with a call to OutputPipe.send() [line 93]:

```
op.send(msg);
```

After sending this message, we close the output pipe and return from this method [line 101]:

```
op.close();
```

rendezvousEvent()

This method [line 109] is called asynchronously whenever we receive a RendezvousEvent. This method is passed one argument:

- RendezvousEvent event — the event that we received from the Rendezvous Service

We expect to receive a connection event (RDVCONNECT) when our peer connects to its rendezvous peer. Other possible events include disconnection events (RDVDISCONNECT), reconnection events (RDVRECONNECT), and rendezvous failure events (RDVFAILED).[1]

When we receive an event of type RendezvousEvent.RDVCONNECT, we notify the thread in the run() method [line 110]:

```
if ( event.getType() == event.RDVCONNECT ) {
   notify();
```

startJxta()

This method instantiates the JXTA platform and creates the default net peer group [line 120]:

```
   netPeerGroup = PeerGroupFactory.newNetPeerGroup();
```

Then it retrieves the Pipe, Discovery, and Rendezvous Services from the default net peer group [line 131]. These services are used later when we create an input pipe:

```
   pipeSvc = netPeerGroup.getPipeService();
   discoverySvc = netPeerGroup.getDiscoveryService();
   rdvSvc = netPeerGroup.getRendezVousService();
```

We then register a rendezvous listener [line 136]:

```
   rdvSvc.addListener( this );
```

Our method rendezvousEvent() will be called whenever we receive an event from the Rendezvous Service.

Lastly, we create a pipe advertisement by reading it in from the existing XML file `examplepipe.adv` [line 141]:

```
   FileInputStream is = new FileInputStream(FILENAME);
```

The file `examplepipe.adv` must exist and it must be valid XML document containing a pipe advertisement, or an exception is raised by the JXTA platform. Recall that the application which creates the input pipe also reads its pipe advertisement from the same file. The contents of the `examplepipe.adv` file are listed in Figure 7-7 on page 71.

1. See class RendezvousEvent for a complete list of possible RendezvousEvents.

As in the previous PipeListener example, the AdvertisementFactory.newAdvertisement() method is called to create a new pipe advertisement [line 142]:

```
        pipeAdv = (PipeAdvertisement)
           AdvertisementFactory.newAdvertisement(
           new MimeMediaType("text/xml"), is);
```

After the pipe advertisement is created, the input stream is closed [line 145] and the method returns:

```
        is.close();
```

Source Code: PipeExample

```java
import java.io.FileInputStream;
import java.io.IOException;
import java.util.Date;

import net.jxta.discovery.DiscoveryService;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.StringMessageElement;
import net.jxta.exception.PeerGroupException;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.OutputPipeEvent;
import net.jxta.pipe.OutputPipeListener;
import net.jxta.pipe.PipeService;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.rendezvous.RendezvousEvent;
import net.jxta.rendezvous.RendezvousListener;
import net.jxta.rendezvous.RendezVousService;

// This example illustrates how to use the OutputPipeListener interface

public class PipeExample implements
    Runnable,
    OutputPipeListener,
    RendezvousListener {

    static PeerGroup netPeerGroup = null;
    private final static String TAG = "PipeListenerMsg";
    private final static String FILENAME = "examplepipe.adv";

    private PipeService pipeSvc;
    private DiscoveryService discoverySvc;
    private RendezVousService rdvSvc;
    private PipeAdvertisement pipeAdv;

    public static void main(String args[]) {
        PipeExample myapp = new PipeExample();
        myapp.startJxta();
        myapp.run();
    }

```

```
     // This thread creates (resolves) the output pipe
     // and sends a message once it's resolved

    public synchronized void run() {
       try {
          //  create output pipe with listener
          //  Send out the first pipe resolve call
          System.out.println("Attempting to create a OutputPipe...");
          pipeSvc.createOutputPipe(pipeAdv, this);

          // send out a second pipe resolution after we connect
          // to a rendezvous
          if ( !rdvSvc.isConnectedToRendezVous() ) {
             System.out.println( "Waiting for Rendezvous Connection" );
             try {
                wait();
                System.out.println( "Connected to Rendezvous, attempting to
create a OutputPipe" );
                pipeSvc.createOutputPipe( pipeAdv, this );
             } catch ( InterruptedException e ) {
                // got our notification
             }
          }

       } catch (IOException e) {
          System.out.println("Error: OutputPipe creation failure");
          e.printStackTrace();
          System.exit(-1);
       }
    }

    // by implementing OutputPipeListener we must define this method
     // that is called when an OutputPipeEvent is received.

    public void outputPipeEvent(OutputPipeEvent event) {

       System.out.println("Got an output pipe event");
       OutputPipe op = event.getOutputPipe();
       Message msg = null;

       // create message, and send it via the output pipe
       try {
          System.out.println( "Sending message" );
          Date date = new Date(System.currentTimeMillis());
          String myMsg = "Hello from peer " + netPeerGroup.getPeerName() +
                         "[" + date.toString() + "]";

          msg = new Message();
          StringMessageElement sme = new StringMessageElement(TAG, myMsg, null);
```

```
            msg.addMessageElement(null, sme);
            op.send(msg);
        }
        catch (Exception e) {
            System.out.println( "Error: failed to send message" );
            e.printStackTrace();
            System.exit(-1);
        }

        op.close();
        System.out.println( "message sent ");
    }


    /**
     * Check if we've received a rendezvous connect event
     */
    public synchronized void rendezvousEvent( RendezvousEvent event ) {
        if ( event.getType() == event.RDVCONNECT ) {
            notify();
        }
    }

    // Starts jxta, and gets the pipe and discovery services

    private void startJxta() {
        try {
            // create and start the default jxta NetPeerGroup
            netPeerGroup = PeerGroupFactory.newNetPeerGroup();

        }
        catch (PeerGroupException e) {
            // could not instantiate the group, print the stack and exit
            System.out.println("fatal error : group creation failure");
            e.printStackTrace();
            System.exit(-1);
        }

        // get the pipe, discovery, and rendezvous services
        pipeSvc = netPeerGroup.getPipeService();
        discoverySvc = netPeerGroup.getDiscoveryService();
        rdvSvc = netPeerGroup.getRendezVousService();

        // Register a rendezvous listener
        rdvSvc.addListener( this );

        // read in pipe advertisement from file
        System.out.println("Reading in from file " + FILENAME);
        try {
            FileInputStream is = new FileInputStream(FILENAME);
            pipeAdv = (PipeAdvertisement)
               AdvertisementFactory.newAdvertisement(
               new MimeMediaType("text/xml"), is);
            is.close();
        }
        catch (Exception e) {
            System.out.println("failed to read/parse pipe advertisement");
            e.printStackTrace();
            System.exit(-1);
        }
    }
}
```

Pipe Advertisement: `examplepipe.adv` **file**

The XML file containing the pipe advertisement, `examplepipe.adv,` is listed in Figure 7-7. This file is read by both the PipeListener and PipeExample classes to create the input and output pipes. Both classes must use the same pipe ID in order to communicate with each other.

```
<!DOCTYPE jxta:PipeAdvertisement>

<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
       <Id>
          urn:jxta:uuid-
59616261646162614A757874614D504725184FBC4E5D498AA0919F662E40028B04
       </Id>
       <Type>
          JxtaUnicast
       </Type>
       <Name>
          PipeExample
       </Name>
</jxta:PipeAdvertisement>
```

Figure 7-7 Pipe advertisement file, `examplepipe.adv`.

Note – Both the PipeListener and PipeExample applications read this file from the current directory. If this file does not exist, or it contains an invalid pipe advertisement, the applications raise an exception and exit.

Using a Bi-Directional Pipe

This example illustrates how to use the bi-directional pipe service to send messages between two JXTA peers. Two separate applications are used in this example:

- BidirectionalAcceptPipeTestApp — creates a bi-directional pipe and waits to receive messages on it
- BidirectionalPipeTestApp — connects to the bi-directional pipe and sends a message on it

Figure 7-8 shows example output when the BidirectionalAcceptPipeTest is run, and Figure 7-9 shows example input from the BidirectionalPipeTestApp:

```
Starting JXTA...
Creating new bps...
Saving advertisement to file: TestPipe.xml
  done.
Waiting for messages:
Sending bidir pipe ack.

BPS: Accepted pipe
net.jxta.impl.util.BidirectionalPipeService$Pipe@18e8541
Received message: Hello, world
  message received at Tue Mar 25 14:06:34 PST 2003
```

Figure 7-8 Example output: BidirectionalAcceptPipeTestApp.

```
Starting JXTA...
Reading in pipe adv. from file: TestPipe.xml
  successfully read pipe advertisement.
Connecting to bps...
  done.
Message sent.
```

Figure 7-9 Example output: BidirectionalPipeTestApp.

Note – If you are running both applications on the same system, you will need to run each one from a separate subdirectory so that they can be configured to use separate ports. The BidirectionalAcceptPipeTestApp application must be run first. It creates a file `TestPipe.xml` which contains the pipe advertisement. After the `TestPipe.xml` file is created, copy it to the subdirectory containing BidirectionalPipeTestApp and run BidirectionalPipeTestApp.

Bidirectional Pipe Service

The BidirectionalPipeService uses the core JXTA uni-directional pipes (InputPipe and OutputPipe) to simulate bi-directional pipes in the J2SE platform binding. The BidirectionalPipeService contains two methods:

- bind — creates a bidirectional pipe that you can use for accepting "connections"
- connect — connects to a bidirectional pipe to send messages

The BidirectionalPipeService also contains two subclasses:

- Pipe — contains methods to retrieve the InputPipe and OutputPipe
- AcceptPipe — contains methods to accept connections, close the pipe, and get the pipe advertisement

The high-level steps used to read from a bi-directional pipe include the following:

- *Bind* — call BidirectionalPipeService.bind(String) and pass it the name of the bi-directional pipe to be created; this returns a BidirectionalPipeService.AcceptPipe object which you can use to accept "connections"
- *Extract pipe advertisement* — if needed, extract the pipe advertisement from the AcceptPipe object using acceptPipe.getAdvertisement(). Any application that wants to connect to you needs to pass this pipe advertisement to BidirectionalPipeService.connect().
- *Accept* — call AcceptPipe.accept(), which accepts the virtual bi-directional pipe connection.
- *Read messages* — call BidirectionalPipeService.Pipe.getInputPipe() to get the input pipe, and then call InputPipe.poll to receive the message. Or, set up a PipeListener to receive notification of InputPipe events asynchronously.

The high-level steps used to write to a bi-directional pipe include the following:

- *Get advertisement* — get the advertisement of the pipe you want to use. The advertisement could be stored in a file, be discovered, or passed to your application.
- *Connect* — call BidirectionalPipeService.connect(PipeAdvertisement); this creates an OutputPipe for our BidirectionalPipeService object and returns a BidirectionalPipeService.Pipe object.
- *Create message* — create a message to send, typically by using your peer group's PipeService.createMessage() method to create a new message and then adding elements with Message.setString(). Both the sender and the receiver must agree upon the element names, or tags, contained in a message.
- *Send the message* — call BidirectionalPipeService.Pipe.getOutputPipe() to get the OutputPipe, and then use OutputPipe.send() to send the message.

The example which reads from a bidirectional pipe, BidirectionalAcceptPipeTestApp, starts on page 73. The example which writes to a bidirectional pipe, BidirectionalPipeTestApp, starts on page 80.

BidirectionalAcceptPipeTestApp

This application creates and listens for messages on a bidirectional pipe. It defines a single class, BidirectionalAcceptPipeTestApp, which implements the Runnable interface. Two class constants contain information about the pipe to be created:

- `String FILENAME` — the XML file in which the pipe advertisement will be stored
- `String TAG` — the message element name, or tag, which we are expecting in any message we receive

We also define the following instance fields:

- `PeerGroup netPeerGroup` — our peer group, the default net peer group
- `BidirectionalPipeService bps` — the pipe service we use to accept connections and receive messages

main()

This method [line 25] creates a new BidirectionalAcceptPipeTestApp object, calls startJxta() to instantiate the JXTA platform and create the default net peer group, and then calls run() which uses the BidirectionalPipeService to accept and receive messages.

startJxta()

This method instantiates the JXTA platform and creates the default net peer group [line 37]:

```
            netPeerGroup = PeerGroupFactory.newNetPeerGroup();
```

run()

This method first creates a new BidirectionalPipeService object [line 52]. It passes the current peer group, netPeerGroup, as an argument:

```
            bps = new BidirectionalPipeService(netPeerGroup);
```

Next, the BidirectionalPipeService.bind() method is called [line 54]. This method takes one argument, which is a string containing the name of the bi-directional pipe to be created:

```
        BidirectionalPipeService.AcceptPipe acceptPipe =
                    bps.bind(pipeName);
```

This method returns a BidirectionalPipeService.AcceptPipe object, which we use later to accept connections and receive incoming messages.

Our method then saves the pipe advertisement to a file [lines 57 to 62]. This pipe advertisement will be used by the partner application, BidirectionalPipeTestApp, enabling both applications to connect to the same bidirectional pipe.

```
            PipeAdvertisement adv = acceptPipe.getAdvertisement();

            // Save the advertisement to a file
            System.out.println("Saving adv to file: " + FILENAME + "...");
            saveAdv(adv, new File(FILENAME));
            System.out.println("  done.");
```

Finally, this method loops forever checking for incoming messages on its bidirectional pipe. It calls accept(), which accepts the virtual bi-directional pipe connection [line 67]:

```
        BidirectionalPipeService.Pipe pipe =
                    acceptPipe.accept(30 * 1000);
```

This method takes one argument, which is the time-out value. It returns a BidirectionalPipeService.Pipe object.

When the accept() method receives an incoming connection, we then extract the InputPipe from the BidirectionalPipeService.Pipe object and call InputPipe.poll() to poll for incoming message [line 72].[1] This method takes one argument, which is the time-out value.

```
        Message msg = pipe.getInputPipe().poll(30 * 1000);
```

If either the accept() or poll() method time-out, an exception is raised but no action is taken. Execution continues in the while loop, and the application loops forever.

1. We could choose to set up a PipeListener to receive notification of InputPipe events asynchronously, rather than polling for events.

saveAdv()

This utility method is used to save an advertisement in a file [line 94]:

```
        static void saveAdv (Advertisement adv, File f) throws IOException {
```

copy()

This utility method is used to copy InputStreams. The first form is used to copy an InputStream to a file [line 111]:

```
        static void copy (InputStream in, File f) throws IOException {
```

The second form is used to copy an InputStream to an OutputStream [line 131]:

```
        static void copy (InputStream in, OutputStream out) throws IOException {
```

Source Code: BidirectionalAcceptPipeTestApp

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Date;
import net.jxta.document.Advertisement;
import net.jxta.document.MimeMediaType;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.MessageElement;
import net.jxta.exception.PeerGroupException;
import net.jxta.impl.util.BidirectionalPipeService;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.protocol.PipeAdvertisement;

public class BidirectionalAcceptPipeTestApp implements Runnable {

   private final static String FILENAME = "TestPipe.xml";
   private final static String TAG = "Test";

   static PeerGroup netPeerGroup = null;
   private BidirectionalPipeService bps;

   public static void main(String args[]) {
      BidirectionalAcceptPipeTestApp myapp =
         new BidirectionalAcceptPipeTestApp();
      myapp.startJxta();
      myapp.run();
   }

   private void startJxta() {

      System.out.println("Starting JXTA...");
      try {
         // create and start the default jxta netPeerGroup
         netPeerGroup = PeerGroupFactory.newNetPeerGroup();
      } catch (PeerGroupException e) {
         // could not instantiate the group; exit
         System.out.println("Fatal error: group creation failure");
         e.printStackTrace();
         System.exit(-1);
      }
    }

    public void run () {
       // the  name of our pipe
       String pipeName = "TestPipe";
       try {
          // create a new BidirectionalPipeService
          System.out.println("Creating new bps...");
          bps = new BidirectionalPipeService(netPeerGroup);

          BidirectionalPipeService.AcceptPipe acceptPipe =
             bps.bind(pipeName);

          PipeAdvertisement adv = acceptPipe.getAdvertisement();

          // Save the advertisement to a file
          System.out.println("Saving adv to file: " + FILENAME + "...");
          saveAdv(adv, new File(FILENAME));
          System.out.println("  done.");

          System.out.println("Waiting for messages:");
          while (true) {
             try {
                BidirectionalPipeService.Pipe pipe =
                   acceptPipe.accept(30 * 1000);
                System.out.println("\nBPS: Accepted pipe " + pipe);

                // extract the message from our input pipe
                Message msg = pipe.getInputPipe().poll(30 * 1000);

                MessageElement el = msg.getMessageElement(null, TAG);
                if (el == null) {
                   System.out.println("Error: could not find element " +
TAG);
                } else {
                   Date date = new Date(System.currentTimeMillis());
                   System.out.println("Received message: " + el.toString());
                   System.out.println("  message received at " +
                                      date.toString());
                }
             } catch (InterruptedException e) {
                    //timed out; continue
             }
          }
       } catch (Exception e) {
          e.printStackTrace();
          System.exit(-1);
       }
    }

    // utilily method; Saves an advertisement in an XML file
    static void saveAdv (Advertisement adv, File f) throws IOException {

       try {
          InputStream in =
             adv.getDocument (new MimeMediaType ("text/xml")).getStream();

          copy (in, f);

       } catch (IOException e) {
          throw e;
       } catch (Exception e) {
          throw new IOException ("Can't get document from adv: " +
             e.getMessage ());
       }
    }

   // utility method; copies an InputStream to a file
    static void copy (InputStream in, File f) throws IOException {

       FileOutputStream out = null;

       try {
          out = new FileOutputStream (f);
          copy (in, out);
       }
       finally {
          if (out != null) {
             try {
                out.close ();
             } catch (IOException e) {
                e.printStackTrace();
             }
          }
       }
     }

    // Utility method; copies an InputStream to an OutputStream
    static void copy (InputStream in, OutputStream out) throws IOException {
       byte[] buf = new byte[8192];
       int r;

       while ((r = in.read (buf)) > 0) {
          out.write (buf, 0, r);
       }
    }
}
```

Example pipe advertisement: TestPipe.xml

An example pipe advertisement, saved to the file `TestPipe.xml`, is listed below:

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

<!DOCTYPE jxta:PipeAdvertisement>

<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
   <Id>
      urn:jxta:uuid-
59616261646162614E504720503250338E3E786229EA460DADC1A176B69B731504
   </Id>
   <Type>
      JxtaUnicast
   </Type>
   <Name>
      TestPipe.end1
   </Name>
</jxta:PipeAdvertisement>
```

BidirectionalPipeTestApp

This example reads in the pipe advertisement that was created by the BidirectionalAcceptPipeTestApp, connects to the bidirectional pipe, and sends a message. It defines a single class, BidirectionalPipeTestApp, that implements the Runnable interface. Two class constants contain information about the pipe to be created:

- `String FILENAME` — The XML file containing the text representation of our pipe advertisement
- `String TAG` — the message element name, or tag, which we must include in any message we send to the BidirectionalAcceptPipeTestApp (the sender and the receiver must agree on the tags used)

We also define the following instance fields:

- `PeerGroup netPeerGroup` — our peer group, the default net peer group
- `PipeService pipeSvc` — the pipe service (from the net peer group)
- `PipeAdvertisement pipeAdv` — the pipe advertisement used in this example
- `BidirectionalPipeService bps` — the pipe service we use to connect to the bi-directional pipe

main()

This method [line 24] creates a new BidirectionalPipeTestApp object, calls startJxta() to instantiate the JXTA platform and create the default net peer group, and then calls run() which uses the BidirectionalPipeService to send a message.

startJxta()

This method instantiates the JXTA platform and creates the default net peer group [line 35]:

```
netPeerGroup = PeerGroupFactory.newNetPeerGroup();
```

It then gets the pipe service from our peer group [line 45]:

```
pipeSvc = netPeerGroup.getPipeService();
```

and creates a new BidirectionalPipeService object [line 48]. It passes the current peer group, netPeerGroup, as an argument:

```
bps = new BidirectionalPipeService(netPeerGroup);
```

Finally, this method reads in the pipe advertisement from a file [lines 53 to 57]:

```
FileInputStream is = new FileInputStream(FILENAME);
pipeAdv = (PipeAdvertisement)
    AdvertisementFactory.newAdvertisement(
    new MimeMediaType("text/xml"), is);
is.close();
```

This file was created by BidirectionalAcceptPipeTestApp. If the file does not exists, or it does not contain a valid pipe advertisement, and exception is raised and this application exits.

run()

This method connects to the same bidirectional pipe as BidirectionalAcceptPipeTestApp and then sends a single message on this pipe.

First, this method calls BidirectionalPipeService.connect() to connect to the bidirectional pipe [line 71]:

```
BidirectionalPipeService.Pipe pipe =
        bps.connect(pipeAdv, 30 * 1000);
```

This method takes two arguments: the pipe advertisement of the pipe to connect to and a time-out value. If the connection succeeds, it returns the corresponding BidirectionalPipeService.Pipe object. If the method is unable to connect in the specified time-out (e.g., the corresponding input pipe hasn't been set up), an exception is raised and the application exits.

Once the connection is established, a new message is created [line 78]:

```
Message msg = new Message();
```

A new StringMessageElement is created with the agreed-upon tag, or element name. The constructor takes three argument: the element tag (or name), the data, and a signature [line 79]:

```
StringMessageElement sme =
         new StringMessageElement(TAG, "Hello, world", null);
```

The new element is added to our message. In this example, we add our element to the null namespace [line 81]:

```
msg.addMessageElement(null, sme);
```

Finally, the OutputPipe is extracted from our BidirectionalPipeService.Pipe object, and the OutputPipe.send() method is used to send our message [line 84]:

```
pipe.getOutputPipe().send(msg);
```

Source Code: BidirectionalPipeTestApp

```
import java.io.FileInputStream;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.StringMessageElement;
import net.jxta.exception.PeerGroupException;
import net.jxta.impl.util.BidirectionalPipeService;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.pipe.PipeService;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.impl.util.BidirectionalPipeService;

public class BidirectionalPipeTestApp   implements Runnable {

    static PeerGroup netPeerGroup = null;
    private final static String FILENAME = "TestPipe.xml";
    private final static String TAG = "Test";

    private PipeService              pipeSvc;
    private PipeAdvertisement        pipeAdv;
    private BidirectionalPipeService bps;

    public static void main(String args[]) {
       BidirectionalPipeTestApp myapp = new BidirectionalPipeTestApp();
       myapp.startJxta();
       myapp.run();
    }

    private void startJxta() {

       System.out.println("Starting JXTA...");
       try {
          // create and start the default jxta netPeerGroup
          netPeerGroup = PeerGroupFactory.newNetPeerGroup();
       }
       catch (PeerGroupException e) {
          // could not instantiate the group; exit
          System.out.println("Fatal error: group creation failure");
          e.printStackTrace();
          System.exit(-1);
       }

       // Get the pipe service from the netPeerGroup
       pipeSvc = netPeerGroup.getPipeService();

```

```
        // create a new BidirectionalPipeService
        bps = new BidirectionalPipeService(netPeerGroup);

        // Read in the pipe advertisement from the file
        System.out.println("Reading in pipe adv. from file: " + FILENAME);
        try{
           FileInputStream is = new FileInputStream(FILENAME);
           pipeAdv = (PipeAdvertisement)
              AdvertisementFactory.newAdvertisement(
              new MimeMediaType("text/xml"), is);
           is.close();
        }
        catch (Exception e) {
           System.out.println("Error: failed to read/parse pipe adv.");
           e.printStackTrace();
           System.exit(-1);
        }
        System.out.println("  successfully read pipe advertisement.");
    }

    public void run() {

        try {
           System.out.println("connecting to bps...");
           BidirectionalPipeService.Pipe pipe =
              bps.connect(pipeAdv, 30 * 1000);

           if (pipe != null) {
              System.out.println("  connected.");

              // Create a new message and add one element
              Message msg = new Message();
              StringMessageElement sme =
                    new StringMessageElement(TAG, "Hello, world", null);
              msg.addMessageElement(null, sme);

              // Send the message
              pipe.getOutputPipe().send(msg);
              System.out.println("Message sent.");
           } else {
                System.out.println("  failed to establish a connection.");
           }
        } catch (Exception e) {
              e.printStackTrace();
              System.exit(-1);
        }
    }
}
```

JXTA Services

JXTA-enabled services are services that are published by a ModuleSpecAdvertisement. A module spec advertisement may include a pipe advertisement that can be used by a peer to create output pipes to invoke the service. Each Jxta-enabled service is uniquely identified by its ModuleSpecID.

There are three separate service-related advertisements:

- *ModuleClassAdvertisement* — defines the service class; its main purpose is to formally document the existence of a module class. It is uniquely identified by a ModuleClassID.
- *ModuleSpecAdvertisement* — defines a service specification; uniquely identified by a ModuleSpecID. Its main purpose is to provide references to the documentation needed in order to create conforming implementations of that specification. A secondary use is to make running instances usable remotely, by publishing any or all of the following:
  - PipeAdvertisement
  - ModuleSpecID of a proxy module
  - ModuleSpecID of an authenticator module
- *ModuleImplAdvertisement* — defines an implementation of a given service specification.

Each of these advertisements serves different purposes, and should be published separately. For example, there are typically more specifications than classes, and more implementations than specifications, and in many cases only the implementation needs to be discovered.

ModuleClassIDs and ModuleSpecIDs are used to uniquely identify the components:

- *ModuleClassID*

  A ModuleClassID uniquely identifies a particular module class. A ModuleClassID is optionally described by a published ModuleClassAdvertisement. It is not required to publish a Module Class Advertisement for a Module Class ID to be valid, although it is a good practice.

- *ModuleSpecID*

  A ModuleSpecID uniquely identifies a particular module specification. Each ModuleSpecID embeds a ModuleClassID which uniquely identifies the base Module class. The specification that corresponds to a given ModuleSpecID may be published in a ModuleSpecAdvertisement. It is not required to publish a Module Spec Advertisement for a ModuleSpecID to be valid, although it is a good practice.

In our example JXTA-enabled service, we create a ModuleClassID and publish it in a ModuleClassAdvertisement. We then create a ModuleSpecID (based on our ModuleClassID) and add it to a ModuleSpecAdvertisement. We then add a pipe advertisement to this ModuleSpecAdvertisement and publish it. Other peers can now discover this ModuleSpecAdvertisement, extract the pipe advertisement, and communicate with our service.

Note – Modules are also used by peer groups to describe peer group services. That discussion is beyond the scope of this example which creates a stand-alone service.

Creating a JXTA Service

This example illustrates how to create a new JXTA service and its service advertisement, publish and search for advertisements via the Discovery service, create a pipe via the Pipe service, and send messages through the pipe. It consists of two separate applications:

- *Server*

  The Server application creates the service advertisements (ModuleClassAdvertisement and ModuleSpecAdvertisement) and publishes them in the NetPeerGroup. The ModuleSpecAdvertisement contains a PipeAdvertisement required to connect to the service. The Server application then starts the service by creating an input pipe to receive messages from clients. The service loops forever, waiting for messages to arrive.

- *Client*

  The Client application discovers the ModuleSpecAdvertisement, extracts the PipeAdvertisement and creates an output pipe to connect to the service, and sends a message to the service.

Figure 7-10 shows example output when the Server application is run, and Figure 7-11 shows example input from the Client application:

```
Starting Service Peer ....
Start the Server daemon
Reading in file pipeserver.adv
Created service advertisement:
jxta:MSA :
        MSID : urn:jxta:uuid-B6F8546BC21D4A8FB47AA68579C9D89EF3670BB315A
C424FA7D1B74079964EA206
        Name : JXTASPEC:JXTA-EX1
        Crtr : sun.com
        SURI : http://www.jxta.org/Ex1
        Vers : Version 1.0
        jxta:PipeAdvertisement :
               Id : urn:jxta:uuid-9CCCDF5AD8154D3D87A391210404E59BE4B888
209A2241A4A162A10916074A9504
                Type : JxtaUnicast
                Name : JXTA-EX1

Waiting for client messages to arrive
Server: received message: Hello my friend!
Waiting for client messages to arrive
```

Figure 7-10 Example output: Server application.

```
Starting Client peer ....
Start the Client
searching for the JXTASPEC:JXTA-EX1 Service advertisement
We found the service advertisement:
jxta:MSA :
        MSID : urn:jxta:uuid-
FDDF532F4AB543C1A1FCBAEE6BC39EFDFE0336E05D31465CBE9
48722030ECAA306
        Name : JXTASPEC:JXTA-EX1
        Crtr : sun.com
        SURI : http://www.jxta.org/Ex1
        Vers : Version 1.0
        jxta:PipeAdvertisement :
                Id : urn:jxta:uuid-
9CCCDF5AD8154D3D87A391210404E59BE4B888209A224
1A4A162A10916074A9504
                Type : JxtaUnicast
                Name : JXTA-EX1

message "Hello my friend!" sent to the Server
Good Bye ....
```

Figure 7-11 Example output: Client application.

Note – If you are running both applications on the same system, you will need to run each one from a separate subdirectory so that they can be configured to use separate ports. The Server application must be run first.

Server

Note – This is the server side of the JXTA-EX1 example. The server side application advertises the JXTA-EX1 service, starts the service, and receives messages on a service-defined pipe endpoint. The service associated module spec and class advertisement are published in the NetPeerGroup. Clients can discover the module advertisements and create output pipes to connect to the service. The server application creates an input pipe that waits to receive messages. Each message received is printed to the screen. We run the server as a daemon in an infinite loop, waiting to receive client messages.

This application defines a single class, Server. Four class constants contain information about the service:

- `String SERVICE` — the name of the service we create and advertise
- `String TAG` — the message element name, or tag, which we are expecting in any message we receive; the client application *must* use this same tag.
- `String NAMESPACE` — the namespace used by the message element; the client application *must* use this same space.
- `String FILENAME` — the name of the file that contains our pipe advertisement. (This file must exist and contain a valid pipe advertisement in order for our application to run correctly.)

We also define several instance fields:

- `PeerGroup group` — our peer group, the default net peer group
- `DiscoveryService discoSvc` — the discovery service; used to publish our new service
- `PipeService pipeSvc` — the pipe service; used to create our input pipe and read messages
- `InputPipe myPipe` — the pipe used to receive messages

main()

This method [line 34] creates a new Server object, calls startJxta() to instantiate the JXTA platform and create the default net peer group, calls startServer() to create and publish the service, and finally calls readMessages() to read messages received by the service.

startJxta()

This method instantiates the JXTA platform and creates the default net peer group [line 47]:

```
group = PeerGroupFactory.newNetPeerGroup();
```

Then it retrieves the discovery and pipe services [line 57]:

```
discoSvc = group.getDiscoveryService();
pipeSvc = group.getPipeService();
```

The discovery service is used later when we publish our service advertisements. The pipe service is used later when we create our input pipe and wait for messages on it.

startServer()

This method creates and publishes the service advertisements. It starts by creating a module class advertisement, which is used to simply advertise the existence of the service. The AdvertisementFactory.newAdvertisement() method is used to create a new advertisement [line 74]:

```
ModuleClassAdvertisement mcadv = (ModuleClassAdvertisement)
             AdvertisementFactory.newAdvertisement(
             ModuleClassAdvertisement.getAdvertisementType());
```

It is passed one argument: the type of advertisement we want to construct. After we create our module class advertisement, we initialize it [lines 78 - 82]:

```
mcadv.setName("JXTAMOD:JXTA-EX1");
mcadv.setDescription("Tutorial example to use JXTA module advertisement
   Framework");

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

The name and description can be any string. A suggested naming convention is to choose a name that starts with "JXTAMOD" to indicate this is a JXTA module. Each module class has a unique ID, which is generated by calling the IDFactory.newModuleClassID() method.

Now that the module class advertisement is created and initialized, it is published in the local cache and propagated to our rendezvous peer [lines 87 - 88]:

```
discoSvc.publish(mcadv, DiscoveryService.ADV);
discoSvc.remotePublish(mcadv, DiscoveryService.ADV);
```

Next, we create the module spec advertisement associated with the service. This advertisement contains all the information necessary for a client to contact the service. For instance, it contains a pipe advertisement to be used to contact the service. Similar to creating the module class advertisement, AdvertisementFactory.newAdvertisement() is used to create a new module spec advertisement [line 98]:

```
ModuleSpecAdvertisement mdadv = (ModuleSpecAdvertisement)
   AdvertisementFactory.newAdvertisement(
   ModuleSpecAdvertisement.getAdvertisementType());
```

After the advertisement is created, we initialize the name, version, creator, ID, and URI [lines 107 - 111]:

```
mdadv.setName(SERVICE);
mdadv.setVersion("Version 1.0");
mdadv.setCreator("sun.com");
mdadv.setModuleSpecID(IDFactory.newModuleSpecID(mcID));
mdadv.setSpecURI("http://www.jxta.org/Ex1");
```

We use IDFactory.newModuleSpecID() to create the ID for our module spec advertisement. This method takes one argument, which is the ID of the associated module class advertisement (created above in line 81).

Note – In practice, you should avoid creating a new ModuleSpecID every time you run your application, because it tends to create many different but equivalent and interchangeable advertisements. This, in turn, would clutter the cache space. It is preferred to allocate a new ModuleSpecID only once, and then hard-code it into your application. A simplistic way to do this is to run your application (or any piece of code) that creates the ModuleSpecID once:

```
IDFactory.newModuleSpecID(mcID)
```

Then print out the resulting ID and use it in your application to recreate the same ID every time:

```
(ModuleSpecID) IDFactory.fromURL(new URL("urn", "",
               "jxta:uuid-<...ID created...>")
```

We now create a new pipe advertisement for our service. The client *must* see use the same advertisement to talk to the server. When the client discovers the module spec advertisement, it will extract the pipe advertisement to create its pipe. We read the pipe advertisement from a default configuration file to ensure that the service will always advertise the same pipe [lines 124 - 128]:

```
        FileInputStream is = new FileInputStream(FILENAME);
        pipeadv = (PipeAdvertisement)
           AdvertisementFactory.newAdvertisement(
              new MimeMediaType("text/xml"), is);
        is.close();
```

After we successfully create our pipe advertisement, we add it to the module spec advertisement [line 136]:

```
        mdadv.setPipeAdvertisement(pipeadv);
```

Now, we have initialized everything we need in our module spec advertisement. We print the complete module spec advertisement as a plain text document [lines 139 - 146], and then we publish it to our local cache and propagate it to our rendezvous peer [lines 150 - 151]:

```
        discoSvc.publish(mdadv, DiscoveryService.ADV);
        discoSvc.remotePublish(mdadv, DiscoveryService.ADV);
```

We're now ready to start the service. We create the input pipe endpoint that clients will use to connect to the service [line 156]:

```
        myPipe = pipeSvc.createInputPipe(pipeadv);
```

readMessages()

This method loops continuously waiting for client messages to arrive on the service's input pipe. It calls PipeService.waitForMessage() [line 176]:

```
        msg = myPipe.waitForMessage();
```

This will block and wait indefinitely until a message is received. When a message is received, we extract the message element with the expected namespace and tag [line 189]:

```
        el = msg.getMessageElement(NAMESPACE, TAG);
```

The Message.getMessageElement() method takes two arguments, a string containing the namespace and a string containing the tag we are looking for. The client and server application *must* agree on the namespace and tag names; this is part of the service protocol defined to access the service.

Finally, assuming we find the expected message element, we print out the message element line 194]:

```
      System.out.println("Server: Received message: " +
                         el.toString());
```

and then continue to wait for the next message.

Source Code: Server

```java
import java.io.InputStream;
import java.io.FileInputStream;
import java.io.StringWriter;

import net.jxta.discovery.DiscoveryService;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredTextDocument;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.MessageElement;
import net.jxta.exception.PeerGroupException;
import net.jxta.id.IDFactory;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.PipeService;
import net.jxta.platform.ModuleClassID;
import net.jxta.protocol.ModuleSpecAdvertisement;
import net.jxta.protocol.ModuleClassAdvertisement;
import net.jxta.protocol.PipeAdvertisement;

public class Server   {

    static PeerGroup group = null;
    private DiscoveryService discoSvc;
    private PipeService pipeSvc;
    private InputPipe myPipe; // input pipe for the service
    private final static String SERVICE = "JXTASPEC:JXTA-EX1"; // service name
    private final static String TAG = "DataTag";                // element tag
    private final static String NAMESPACE = "myService";        // element
namespace
    private final static String FILENAME = "pipeserver.adv";   // file containing
                                                                // pipe adv.

    public static void main(String args[]) {
        Server myapp = new Server();
        System.out.println("Starting Service Peer ....");
        myapp.startJxta();
        myapp.startServer();
        myapp.readMessages();
        System.out.println("Good Bye ....");
        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 : group creation failure");
            e.printStackTrace();
            System.exit(1);
        }

        // get the discovery and pipe services
        discoSvc = group.getDiscoveryService();
        pipeSvc = group.getPipeService();
    }

    private void startServer() {

        System.out.println("Start the Server daemon");

        try {

            // Create the Module class advertisement associated with the service
            // We build the module class advertisement using the Advertisement
            // Factory class by passing it the type of the advertisement we
            // want to construct. The Module class advertisement is a
            // a very small advertisement that only advertises the existence
            // of service. In order to access the service, a peer will
            // have to discover the associated module spec advertisement.
            ModuleClassAdvertisement mcadv = (ModuleClassAdvertisement)
                AdvertisementFactory.newAdvertisement(
                ModuleClassAdvertisement.getAdvertisementType());

            mcadv.setName("JXTAMOD:JXTA-EX1");
            mcadv.setDescription("Tutorial example to use JXTA module
advertisement Framework");

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

            // Ok the Module Class advertisement was created, just publish
            // it in my local cache and to my peergroup. This
            // is the NetPeerGroup
            discoSvc.publish(mcadv, DiscoveryService.ADV);
            discoSvc.remotePublish(mcadv, DiscoveryService.ADV);

            // Create the Module Spec advertisement associated with the service
            // We build the module Spec Advertisement using the advertisement
            // Factory class by passing in the type of the advertisement we
            // want to construct. The Module Spec advertisement will contain
            // all the information necessary for a client to contact the service
            // for instance it will contain a pipe advertisement to
            // be used to contact the service

            ModuleSpecAdvertisement mdadv = (ModuleSpecAdvertisement)
                AdvertisementFactory.newAdvertisement(
                ModuleSpecAdvertisement.getAdvertisementType());

            // Setup some of the information field about the servive. In this
            // example, we just set the name, provider and version and a pipe
            // advertisement. The module creates an input pipes to listen
            // on this pipe endpoint.

            mdadv.setName(SERVICE);
            mdadv.setVersion("Version 1.0");
            mdadv.setCreator("sun.com");
            mdadv.setModuleSpecID(IDFactory.newModuleSpecID(mcID));
            mdadv.setSpecURI("http://www.jxta.org/Ex1");

            // Create a pipe advertisement for the Service. The client MUST use
            // the same pipe advertisement to talk to the server. When the client
            // discovers the module advertisement it will extract the pipe
            // advertisement to create its pipe. So, we are reading the pipe
            // advertisement from a default config file to ensure that the
            // service will always advertise the same pipefs

            System.out.println("Reading in file " + FILENAME);
            PipeAdvertisement pipeadv = null;

            try {
                FileInputStream is = new FileInputStream(FILENAME);
                pipeadv = (PipeAdvertisement)
                    AdvertisementFactory.newAdvertisement(
                    new MimeMediaType("text/xml"), is);
                is.close();
            } catch (java.io.IOException e) {
                System.out.println("failed to read/parse pipe advertisement");
                e.printStackTrace();
                System.exit(-1);
            }

            // add the pipe advertisement to the ModuleSpecAdvertisement
            mdadv.setPipeAdvertisement(pipeadv);

```

```
            // display the advertisement as a plain text document.
            System.out.println("Created service advertisement:");
            StructuredTextDocument doc = (StructuredTextDocument)
            mdadv.getDocument(new MimeMediaType("text/plain"));

            StringWriter out = new StringWriter();
            doc.sendToWriter(out);
            System.out.println(out.toString());
            out.close();

            // Ok the Module advertisement was created, just publish
            // it in my local cache and into the NetPeerGroup.
            discoSvc.publish(mdadv, DiscoveryService.ADV);
            discoSvc.remotePublish(mdadv, DiscoveryService.ADV);

            // We are now ready to start the service --
            // create the input pipe endpoint clients will
            // use to connect to the service
            myPipe = pipeSvc.createInputPipe(pipeadv);

        } catch (Exception ex) {
            ex.printStackTrace();
            System.out.println("Server: Error publishing the module");
        }
    }

    private void readMessages() {
        Message msg = null;
        MessageElement el = null;

    // Loop over every input received from client (OK, no way to
    // stop this daemon, but that's beyond the point of the example)
    while (true) {

            System.out.println("Waiting for client messages to arrive");

            try {
                // Listen on the pipe for a client message
                msg = myPipe.waitForMessage();

            } catch (Exception e) {
                myPipe.close();
                System.out.println("Server: Error listening for message");
                return;
            }

            if (msg == null) {
                System.out.println("Null message received.");
            }
            else {
               // Get message element with our NAMESPACE:TAG
               el = msg.getMessageElement(NAMESPACE, TAG);
               if (el == null)
                  System.out.println("Server: error, could not find element " +
                                     NAMESPACE + ":" + TAG);
               else
                  System.out.println("Server: Received message: " +
                                     el.toString());
               }
            }
         }
      }
```

Example Service Advertisement: `pipeserver.adv` **file**

An example pipe advertisement, stored in the `pipeserver.adv` file, is listed below:

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

<!DOCTYPE jxta:PipeAdvertisement>

<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
   <Id>
      urn:jxta:uuid-
9CCCDF5AD8154D3D87A391210404E59BE4B888209A2241A4A162A10916074A9504
   </Id>
   <Type>
      JxtaUnicast
   </Type>
   <Name>
JXTA-EX1
   </Name>
</jxta:PipeAdvertisement>
```

Client

Note – This is the client side of the EX1 example that looks for the JXTA-EX1 service and connects to its advertised pipe. The Service advertisement is published in the NetPeerGroup by the server application. The client discovers the service advertisement and creates an output pipe to connect to the service input pipe. The server application creates an input pipe that waits to receive messages. Each message receive is displayed to the screen. The client sends an hello message.

This application defines a single class, Client. Three class constants contain information about the service:

- `String SERVICE` — the name of the service we are looking for (advertised by Server)
- `String TAG` — the message element name, or tag, which we include in any message we send; the Server application *must* use this same tag.
- `String NAMESPACE` — the namespace used by the message element; the client application *must* use this same space.

We also define several instance fields:

- `PeerGroup netPeerGroup` — our peer group, the default net peer group
- `DiscoveryService discoSvc` — the discovery service; used to find the service
- `PipeService pipeSvc` — the pipe service; used to create our output pipe and send messages

main()

This method [line 30] creates a new Client object, calls startJxta() to instantiate the JXTA platform and create the default net peer group, calls startClient() to find the service and send a messages.

startJxta()

This method instantiates the JXTA platform and creates the default net peer group [line 42]:

```
group = PeerGroupFactory.newNetPeerGroup();
```

Then it retrieves the discovery and pipe services [line 51]:

```
discoSvc = group.getDiscoveryService();
pipeSvc = group.getPipeService();
```

The discovery service is used later when we look for the service advertisement. The pipe service is used later when we create our output pipe and send a message on it.

startClient()

This method loops until it locates the service advertisement. It first looks in the local cache to see if it can discover an advertisement which has the (Name, JXTASPEC: JXTA-EX1) tag and value pair [line 73]:

```
enum = discoSvc.getLocalAdvertisements(DiscoveryService.ADV,
                                       "Name",
                                       SERVICE);
```

We pass the DiscoveryService.getLocalAdvertisements() method three arguments: the type of advertisement we're looking for, the tag ("Name"), and the value for that tag. This method returns an enumeration of all advertisements that exactly match this tag/value pair; if no matching advertisements are found, this method returns null.

If we don't find the advertisement in our local cache, we send a remote discovery request searching for the service [line 82]:

```
        discoSvc.getRemoteAdvertisements(null,
                                         DiscoveryService.ADV,
                                         "Name",
                                         SERVICE,
                                         1,
                                          null);
```

We pass the DiscoveryService.getRemoteAdvertisements() method 6 arguments:

- `java.lang.string peerid` — id of a peer to connect to; if null, connect to rendezvous peer
- `int type` — PEER, GROUP, ADV
- `java.lang.string attribute` — attribute name to narrow discovery to
- `java.lang.string value` — value of attribute to narrow discovery to
- `int threshold` — the upper limit of responses from one peer
- `net.jxta.discovery.DiscoveryListener listener` — discovery listener service to be used

Since discovery is asynchronous, we don't know how long it will take. We sleep as long as we want, and then try again.

When a matching advertisement is found, we break from the loop and continue on. We retrieve the module spec advertisement from the enumeration of advertisements that were found [line 104]:

```
        ModuleSpecAdvertisement mdsadv = (ModuleSpecAdvertisement)
     enum.nextElement();
```

We print the advertisement as a plain text document [lines 109 - 115] and then extract the pipe advertisement from the module spec advertisement [line 118]:

```
        PipeAdvertisement pipeadv = mdsadv.getPipeAdvertisement();
```

Now that we have the pipe advertisement, we can create an output pipe and use it to connect to the server. In our example, we try three times to bind the pipe endpoint to the listening endpoint pipe of the service using PipeService.createOutputPipe() [line 133]:

```
        myPipe = pipeSvc.createOutputPipe(pipeadv, 10000);
```

Next, we create a new (empty) message and a new element. The element contains the agreed-upon element tag, the data (our message to send), and a null signature [line 147]:

```
        Message msg = new Message();
        StringMessageElement sme = new StringMessageElement(TAG, data, null);
```

We add the element to our message in the agreed-upon namespace:

```
        msg.addMessageElement(NAMESPACE, sme);
```

The only thing left to do is send the message to the service using the PipeService.send() method [line 152]:

```
        myPipe.send(msg);
```

Source Code: Client

```
import java.io.IOException;
import java.io.StringWriter;
import java.util.Enumeration;

import net.jxta.discovery.DiscoveryService;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredTextDocument;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.StringMessageElement;
import net.jxta.exception.PeerGroupException;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.PipeService;
import net.jxta.protocol.ModuleSpecAdvertisement;
import net.jxta.protocol.PipeAdvertisement;

public class Client {

    static PeerGroup netPeerGroup = null;
    private DiscoveryService discoSvc;
    private PipeService pipeSvc;

    private OutputPipe myPipe; // Output pipe to connect the service
    private final static String SERVICE = "JXTASPEC:JXTA-EX1";// service name
    private final static String TAG = "DataTag";          // element tag
   private final static String NAMESPACE = "myService";     // element namespace

    public static void main(String args[]) {
        Client myapp = new Client();
        System.out.println("Starting Client peer ....");
        myapp.startJxta();
        myapp.startClient();
        System.out.println("Good Bye ....");
        System.exit(0);
    }

    private void startJxta() {
        try {
            // create, and Start the default jxta NetPeerGroup
            netPeerGroup = PeerGroupFactory.newNetPeerGroup();
        } catch (PeerGroupException e) {
            // could not instantiate the group, print the stack and exit
            System.out.println("fatal error : group creation failure");
            e.printStackTrace();
            System.exit(1);
            }

            // get the discovery and pipe services
            discoSvc = netPeerGroup.getDiscoveryService();
            pipeSvc = netPeerGroup.getPipeService();

        }

       // start the client
      private void startClient() {

          System.out.println("Start the Client");

          // Let's try to locate the service advertisement SERVICE
          // we will loop until we find it!
          System.out.println("searching for the " + SERVICE +
                             " Service advertisement");
          Enumeration enum = null;
          while (true) {
              try {

                  // let's look first in our local cache to see
                  // if we have it. We try to discover an advertisement
                  // which has the (Name, JXTASPEC:JXTA-EX1) tag value

                  enum = discoSvc.getLocalAdvertisements(DiscoveryService.ADV,
                                                         "Name",
                                                         SERVICE);

                  // Found it! Stop searching and go send a message.
                  if ((enum != null) && enum.hasMoreElements()) break;

                  // We could not find anything in our local cache, so let's send a
                  // remote discovery request searching for the service adv
                  discoSvc.getRemoteAdvertisements(null,
                                                   DiscoveryService.ADV,
                                                   "Name",
                                                   SERVICE,1, null);

                  // The discovery is asynchronous and we do not know
                  // how long is going to take (could implement asynchronous
                  // listener pipe)
                  try {
                      Thread.sleep(2000);
                  } catch (Exception e){}

                } catch (IOException e){
                    // found nothing -- move on
                }

                System.out.print(".");
            }

            System.out.println("We found the service advertisement:");

            // Ok get the service advertisement as a ModuleSpecAdvertisement
            ModuleSpecAdvertisement mdsadv =
              (ModuleSpecAdvertisement) enum.nextElement();
          try {

              // let’s print the advertisement as a plain text document
              StructuredTextDocument doc = (StructuredTextDocument)
              mdsadv.getDocument(new MimeMediaType("text/plain"));

              StringWriter out = new StringWriter();
              doc.sendToWriter(out);
              System.out.println(out.toString());
              out.close();

              // Get the pipe advertisement -- need it to talk to the service
              PipeAdvertisement pipeadv = mdsadv.getPipeAdvertisement();

              if (pipeadv == null){
                  System.out.println("Error -- Null pipe advertisement!");
                  System.exit(1);
              }

              // create the output pipe endpoint to connect
              // to the server, try 3 times to bind the pipe endpoint to
              // the listening endpoint pipe of the service
              myPipe = null;
              for (int i=0; i<3; i++) {

                  System.out.println("Trying to bind to pipe...");
                  try {
                      myPipe = pipeSvc.createOutputPipe(pipeadv, 10000);
                      break;
                  } catch (java.io.IOException e) {
                      // go try again;
                  }
              }
              if (myPipe == null) {
                  System.out.println("Error resolving pipe endpoint");
                  System.exit(1);
              }
              // create the data string to send to the server
              String data = "Hello my friend!";

```

```
            // create the pipe message
            Message msg = new Message();
            StringMessageElement sme = new StringMessageElement(TAG, data,
null);
            msg.addMessageElement(NAMESPACE, sme);

            // send the message to the service pipe
            myPipe.send(msg);
            System.out.println("message \"" + data + "\" sent to the Server");

        } catch (Exception ex) {
            ex.printStackTrace();
            System.out.println("Client: Error sending message to the service");
        }
    }
}
```

Creating a Secure Peer Group

This example[1] illustrates how to create and join a new peer group that implements authentication via a login and a password.

Figure 7-12 shows example output when this application is run:

```
JXTA platform Started ...
 Peer Group Created ...
 Peer Group Found ...
 Peer Group Joined ...
--------------------------------------------------
| XML Advertisement for Peer Group Advertisement |
--------------------------------------------------
<?xml version="1.0"?>

<!DOCTYPE jxta:PGA>

<jxta:PGA xmlns:jxta="http://jxta.org">
        <GID>
                urn:jxta:uuid-4D6172676572696E204272756E6F202002
        </GID>
        <MSID>
                urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000010406
        </MSID>
        <Name>
                SatellaGroup
        </Name>
        <Desc>
                Peer Group using Password Authentication
        </Desc>
        <Svc>
                <MCID>
                      urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000505
                </MCID>
                <Parm>
                        <login>
                                SecurePeerGroups:FZUH:
                        </login>
                </Parm>
        </Svc>
</jxta:PGA>
--------------------------------------------------
```

Figure 7-12 Example output: Creating and joining a peer group that requires authentication.

1. This example was provided by Bruno Margerin of Science System & Applications, Inc. Portions of the code were taken from the Instant P2P and JXTA Shell projects.

Note – The password encryption used in net.jxta.impl.membership.PasswdMembershipService is extremely weak and has been cracked over 2 millenniums ago. So, this method is highly unsecure. But the principle for joining a group with better password encryption method remains the same.

You can also join the authenticated peer group with other JXTA applications, such as the JXTA shell, using the following user and password:

- Login: SecurePeerGroups
- Password: RULE

main()

This method call the constructor SecurePeerGroup() of the class and instantiates a new SecurePeerGroup Object called satellaRoot.

The constructor method SecurePeerGroup()

This method creates and joins the secure peer group, and then prints the peer group's advertisement. More specifically, this method:

- Instantiates the JXTA platform and creates the default netPeerGroup by calling the startJxta() method [line 44]
- Instantiates the user login, password, group name and group ID [lines 54-71]
- Creates the authenticated peer group called "SatellaGroup" by calling the createPeerGroup() method [line 75]
- Searches for the "SatellaGroup" peer group by calling the discoverPeerGroup() method [line 80]
- Joins the "SatellaGroup" peer group by calling the joinPeerGroup() method [line 84]
- Prints on standard output the XML Advertisement of the "SatellaGroup" peer group by calling the printXmlAdvertisement() method [line 89]

StartJxta()

This method instantiates the JXTA platform, creates (and later returns) the default netPeerGroup called myNetPeerGroup [line 485]:

```
myNetPeerGroup=PeerGroupFactory.newNetPeerGroup();
```

createPeerGroup()

The peer group that is being built does not have the same characteristics than the standard peer group. Indeed, it has a different membership implementation: it uses the net.jxta.impl.membership.PasswdMembershipService instead of the regular net.jxta.impl.membership.NullMembershipService. Therefore, it is required to create and publish a new Peer Group Module Implementation that reflects this new implementation of the Membership Service [line 105]:

```
passwdMembershipModuleImplAdv=
   this.createPasswdMembershipPeerGroupModuleImplAdv(rootPeerGroup);
```

Once created, this advertisement is published locally and remotely in the parent group using the parent peer group's Discovery Service [line 111]:

```
rootPeerGroupDiscoveryService.publish(passwdMembershipModuleImplAdv,
                                      DiscoveryService.ADV,
                                      PeerGroup.DEFAULT_LIFETIME,
                                      PeerGroup.DEFAULT_EXPIRATION);
rootPeerGroupDiscoveryService.remotePublish(passwdMembershipModuleImplAdv,
                                            DiscoveryService.ADV,
                                            PeerGroup.DEFAULT_EXPIRATION);
```

Once this Peer Group Module Implementation in created and published, the createPeerGroup() method binds the new Module Implementation advertisement, peer group name, login and password together into the actual Peer Group advertisement by calling the createPeerGroupAdvertisement() method, [line 123]:

```
satellaPeerGroupAdv=
   this.createPeerGroupAdvertisement(passwdMembershipModuleImplAdv,
      groupName,login,passwd);
```

And publishes it locally and remotely in the parent group using the parent peer group's Discovery Service [line 130-135]:

```
rootPeerGroupDiscoveryService.publish(satellaPeerGroupAdv,
                                      DiscoveryService.GROUP,
                                      PeerGroup.DEFAULT_LIFETIME,
                                      PeerGroup.DEFAULT_EXPIRATION);
rootPeerGroupDiscoveryService.remotePublish(satellaPeerGroupAdv,
                                            DiscoveryService.GROUP,
                                            PeerGroup.DEFAULT_EXPIRATION);
```

Finally the peer group is created from the peer group advertisement [line 145]:

```
satellaPeerGroup=rootPeerGroup.newGroup(satellaPeerGroupAdv);
```

And returned [line 151]:

```
return satellaPeerGroup;
```

createPasswdMembershipPeerGroupModuleImplAdv ()

This method creates the module implementation advertisement for the peer group. It relies on a second method createPasswdMembershipServiceModuleImplAdv () for creating the module implementation advertisement for the membership service.

This method relies on generic, standard "allPurpose" Advertisements that it modifies to take into account the new membership implementation. (Appendix E contains a typical All Purpose Peer Group Module Implementation Advertisement for your reference.)

You can see that the "Param" Element contains all the peer group services including the Membership Service (see Figure 7-13). Therefore, most of the work will be performed of this piece of the document.

The following tasks are performed:

- Create a standard generic peer group module implementation advertisement [line 207]:

```
allPurposePeerGroupImplAdv=
   rootPeerGroup.getAllPurposePeerGroupImplAdvertisement();
```

- Extract its "Param" element. As mentioned above, this contains the services provided by the peer group [line 219]:
```
passwdMembershipPeerGroupParamAdv =
    new StdPeerGroupParamAdv(allPurposePeerGroupImplAdv.getParam());
```

- From this "Param" element, extract the peer group services and their associated service IDs [line 226-229]:
```
Hashtable allPurposePeerGroupServicesHashtable=
    passwdMembershipPeerGroupParamAdv.getServices();
Enumeration allPurposePeerGroupServicesEnumeration=
    allPurposePeerGroupServicesHashtable.keys();
```
- Loop through all this services looking for the Membership Services. The search is performed by looking for the ID matching the MembershipService ID [line 235]:
```
if (allPurposePeerGroupServiceID.equals(PeerGroup.membershipClassID))
```
- And once found, extract the generic membership service advertisement [line 239]:
```
ModuleImplAdvertisement allPurposePeerGroupMemershipServiceModuleImplAdv=
    (ModuleImplAdvertisement)
    allPurposePeerGroupServicesHashtable.get(allPurposePeerGroupServiceID);
```
- Use this generic advertisement to generate a custom one for the Password Membership Service using the createPasswdMembershipServiceModuleImplAdv() method [line 243]:
```
passwdMembershipServiceModuleImplAdv=
    this.createPasswdMembershipServiceModuleImplAdv
    (allPurposePeerGroupMemershipServiceModuleImplAdv);
```
- Remove the generic Membership advertisement [line 247]:
```
allPurposePeerGroupServicesHashtable.remove(allPurposePeerGroupServiceID);
```
- And replace it by the new one [line 249]:
```
allPurposePeerGroupServicesHashtable.put
    (PeerGroup.membershipClassID,passwdMembershipServiceModuleImplAdv);
```
- Finally replace the "Param" element that has just been updated with the new PasswdMembershipService in the peer group module implementation [line 255]:
```
passwdMembershipPeerGroupModuleImplAdv.setParam(
    (Element)PasswdMembershipPeerGroupParamAdv.getDocument(new
       MimeMediaType("text/xml")));
```
- And Update the Password Membership peer group module implementation advertisement spec ID [line 266]. Since the new Peer group module implementation advertisement is no longer the "AllPurpose" one, it should therefore not refer to the "allPurpose" peer group spec advertisement:
```
passwdGrpModSpecID = IDFactory.fromURL(new URL("urn","",
    "jxta:uuid-"+"DeadBeefDeafBabaFeedBabe00000001" +"04" +"06" ) );

passwdMembershipPeerGroupModuleImplAdv.setModuleSpecID(
    (ModuleSpecID) passwdGrpModSpecID);
```

CreatePasswdMembershipServiceModuleImplAdv()

This method works like the previous one: it takes a generic advertisement and uses it to create a customized one.

Figure 7-13 lists the generic advertisement that is receives as argument by this method.

```
<Svc>
    <jxta:MIA>
        <MSID>
            urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000050106
        </MSID>
        <Comp>
            <Efmt>
                JDK1.4
            </Efmt>
            <Bind>
                V1.0 Ref Impl
            </Bind>
        </Comp>
        <Code>
            net.jxta.impl.membership.NullMembershipService
        </Code>
        <PURI>
            http://www.jxta.org/download/jxta.jar
        </PURI>
        <Prov>
            sun.com
        </Prov>
        <Desc>
            Reference Implementation of the MembershipService service
        </Desc>
    </jxta:MIA>
</Svc>
```

Figure 7-13 XML representation of a typical MembershipService, extracted from the Parm element of a peer group module implementation advertisement.

This method needs only to update the Module Spec ID, the code, and description with values specific to the Passwd-MembershipService [lines 288]:

```
passwdMembershipServiceModuleImplAdv.setModuleSpecID(
   PasswdMembershipService.passwordMembershipSpecID);

passwdMembershipServiceModuleImplAdv.setCode(
   PasswdMembershipService.class.getName());

passwdMembershipServiceModuleImplAdv.setDescription(
   "Module Impl Advertisement for the PasswdMembership Service");
```

The rest of the PasswdMembershipServiceAdvertisement is just plain copies of the generic one [line 294]:

```
passwdMembershipServiceModuleImplAdv.setCompat(
   allPurposePeerGroupMemershipServiceModuleImplAdv.getCompat());
```

```
passwdMembershipServiceModuleImplAdv.setUri(
   allPurposePeerGroupMemershipServiceModuleImplAdv.getUri());

passwdMembershipServiceModuleImplAdv.setProvider(
   allPurposePeerGroupMemershipServiceModuleImplAdv.getProvider());
```

createPeerGroupAdvertisement()

This methods creates peer group advertisement from scratch using the advertisement factory [line 159]:

```
PeerGroupAdvertisement satellaPeerGroupAdv=
   (PeerGroupAdvertisement)
      AdvertisementFactory.newAdvertisement(
         PeerGroupAdvertisement.getAdvertisementType());
```

And initializes the specifics of this instance of our authenticated peer group. That is:

- Its peer group ID. In this example, the peer group ID is fixed, so that each time the platform is started the same peer group is created [line 167]:
  ```
  satellaPeerGroupAdv.setPeerGroupID(satellaPeerGroupID);
  ```
- Its Module Spec ID advertisement from which the peer group will find which peer group implementation to use. In this example, this implementation is the Password Membership Module Implementation [line 168]
  ```
  satellaPeerGroupAdv.setModuleSpecID(
  passwdMembershipModuleImplAdv.getModuleSpecID());
  ```
- Its name and description [line 170]:
  ```
  satellaPeerGroupAdv.setName(groupName);

  satellaPeerGroupAdv.setDescription(
     "Peer Group using Password Authentication");
  ```

User and password information is structured as a "login" XML Element and is included into the XML document describing the Service Parameters of the Peer group.

Line 176 shows the creation of this Service Parameters XML document:

```
StructuredTextDocument loginAndPasswd= (StructuredTextDocument)
StructuredDocumentFactory.newStructuredDocument(new MimeMediaType
   ("text/xml"),"Parm");
```

Whereas lines 180-183 show the creation of the "login" XML Element:

```
String loginAndPasswdString =
   login + ":" + PasswdMembershipService.makePsswd(passwd) + ":";
TextElement loginElement =
   loginAndPasswd.createElement("login",loginAndPasswdString);
```

discoverPeerGroup()

This method extracts the discovery service from the parent group (netpeergroup, in our example) [line 313]:

```
myNetPeerGroupDiscoveryService = myNetPeerGroup.getDiscoveryService();
```

And uses this service to look for the newly created peer group ("SatellaGroup") advertisement in the local cache. The search is conducted by looking for peer group advertisements whose peer group ID matches the "SatellaGroup" one.

The method loops until it finds it. Since we published the peer group Advertisement locally we know it is there, and therefore there is no need to remote query the P2P network [line 325]:

```
localPeerGroupAdvertisementEnumeration=
   myNetPeerGroupDiscoveryService.getLocalAdvertisements(
      DiscoveryService.GROUP,"GID",satellaPeerGroupID.toString());
```

Once the correct peer group advertisement is found, the corresponding peer group is created using the parent group (here, netPeerGroup) newgroup() method [line 349]:

```
satellaPeerGroup=myNetPeerGroup.newGroup(satellaPeerGroupAdv);
```

joinPeerGroup()

This method is very similar to the joinGroup() method described earlier (see “Joining a Peer Group” on page 50). It uses the same "apply" and "join" steps. But, unlike the nullAuthenticationService where there is no authentication to complete, the PasswdAuthenticationService requires some authentication. It essentially resides in providing a user login and a password [line 380]:

```
completeAuth( auth, login, passwd );
```

completeAuth()

This method performs the authentication completion required before being able to join the peer group. In orders to complete the authentication, the authentication methods needs to be extracted from the Authenticator. These method’s name starts with "setAuth". Specifically the "setAuth1Identity" method need to be provided with the correct login and "setAuth2_Password" with the correct password.

The methods are extracted from the Authenticator [line 403]:

```
Method [] methods = auth.getClass().getMethods();
```

Then the Authenticator method are filtered and sorted and placed into a Vector, keeping only the ones that are relevant to the authentication process (starting with "setAuth") [line 409 -432]

And goes through all the Authentication method place into looking for "setAuth1Identity" and "setAuth2_Password" and invokes them with the appropriate parameters: [Line 434-452]

```
   Object [] AuthId = {login};
   Object [] AuthPasswd = {passwd};

   for( int eachAuthMethod=0;eachAuthMethod<authMethods.size();
   eachAuthMethod++ ) {
      Method doingMethod = (Method) authMethods.elementAt(eachAuthMethod);

      String authStepName = doingMethod.getName().substring(7);
      if (doingMethod.getName().equals("setAuth1Identity")) {
         // Found identity Method, providing identity
         doingMethod.invoke( auth, AuthId);
      }
      else if (doingMethod.getName().equals("setAuth2_Password")){
         // Found Passwd Method, providing passwd
         doingMethod.invoke( auth, AuthPasswd );
      }
   }
}
```

Source Code: SecurePeerGroup

```java
import java.io.StringWriter;
import java.lang.reflect.Method;
import java.lang.reflect.Modifier;
import java.net.URL;
import java.util.Enumeration;
import java.util.Hashtable;
import java.util.Vector;
import net.jxta.credential.AuthenticationCredential;
import net.jxta.discovery.DiscoveryService;
import net.jxta.document.Advertisement;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredTextDocument;
import net.jxta.document.TextElement;
import net.jxta.endpoint.*;
import net.jxta.exception.PeerGroupException;
import net.jxta.id.ID;
import net.jxta.id.IDFactory;
import net.jxta.impl.membership.PasswdMembershipService;
import net.jxta.impl.protocol.*;
import net.jxta.membership.Authenticator;
import net.jxta.membership.MembershipService;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.platform.ModuleSpecID;
import net.jxta.protocol.ModuleImplAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;

public class SecurePeerGroup  {

    private PeerGroup  myNetPeerGroup=null,
        satellaPeerGroup=null,discoveredSatellaPeerGroup=null;
    private static PeerGroupID satellaPeerGroupID;
    private final static String GROUPID =
        "jxta:uuid-4d6172676572696e204272756e6f202002";

    /** Creates new RootWS */
    public SecurePeerGroup() {
        // Starts the JXTA Platform
        myNetPeerGroup=this.startJxta();
        if (myNetPeerGroup!=null){
            System.out.println("JXTA platform Started ...");
        } else {
           System.err.println(" JXTA plateform has failed to start:  myNetPeerGroup is null");
            System.exit(1);
        }
        //Generate the parameters:
        // login, passwd, peer group name and peer group ID
        // for creating the Peer Group
        String login="SecurePeerGroups";
        String passwd="RULE";
        String groupName="SatellaGroup";

        // the peer group ID is constant so that the same peer group is
        //recreated each time.
        try{
            satellaPeerGroupID = (PeerGroupID) net.jxta.id.IDFactory.fromURL(
               new java.net.URL("urn","", GROUPID));
        }
        catch (java.net.MalformedURLException e) {
         System.err.println(" Can't create satellaPeerGroupID:  MalformedURLException");

            System.exit(1);
        }
        catch (java.net.UnknownServiceException e) {
         System.err.println(" Can't create satellaPeerGroupID: UnknownServiceException");

            System.exit(1);
        }

        // create The Passwd Authenticated Peer Group
        satellaPeerGroup =
           this.createPeerGroup(myNetPeerGroup,groupName,login,passwd);

        // join the satellaPeerGroup
        if (satellaPeerGroup!=null){
            System.out.println(" Peer Group Created ...");
            discoveredSatellaPeerGroup=
               this.discoverPeerGroup(myNetPeerGroup,satellaPeerGroupID);
            if (discoveredSatellaPeerGroup!=null){
                System.out.println(" Peer Group Found ...");
                this.joinPeerGroup(discoveredSatellaPeerGroup, login, passwd);
            }
        }
        System.out.println(" Peer Group Joined ...");
        // Print the Peer Group Adverstisement on sdt out.
        this.printXmlAdvertisement("XML Advertisement for Peer Group Advertisement",
            satellaPeerGroup.getPeerGroupAdvertisement() );
    }

    private PeerGroup createPeerGroup( PeerGroup rootPeerGroup,String groupName,
                       String login, String passwd ) {

        // create the Peer Group by doing the following:
        // - Create a Peer Group Module Implementation Advertisement and publish it
        // - Create a Peer Group Adv and publish it
        // - Create a Peer Group from the Peer Group Adv and return this object
        PeerGroup satellaPeerGroup=null;
        PeerGroupAdvertisement satellaPeerGroupAdv;

        // Create the PeerGroup Module Implementation Adv
        ModuleImplAdvertisement passwdMembershipModuleImplAdv ;
        passwdMembershipModuleImplAdv=
           this.createPasswdMembershipPeerGroupModuleImplAdv(rootPeerGroup);
        // Publish it in the parent peer group
        DiscoveryService rootPeerGroupDiscoveryService =
            rootPeerGroup.getDiscoveryService();
        try {
            rootPeerGroupDiscoveryService.publish(passwdMembershipModuleImplAdv,
                                                  DiscoveryService.ADV,
                                                  PeerGroup.DEFAULT_LIFETIME,
                                                  PeerGroup.DEFAULT_EXPIRATION);
            rootPeerGroupDiscoveryService.remotePublish(passwdMembershipModuleImplAdv,
                                                        DiscoveryService.ADV,
                                                        PeerGroup.DEFAULT_EXPIRATION);
        } catch (java.io.IOException e) {
            System.err.println("Can't Publish passwdMembershipModuleImplAdv");
            System.exit(1);
        }
        // Now, Create the Peer Group Advertisement
        satellaPeerGroupAdv=
           this.createPeerGroupAdvertisement(passwdMembershipModuleImplAdv,
                                             groupName,
                                             login,
                                             passwd);
        // Publish it in the parent peer group
        try {
            rootPeerGroupDiscoveryService.publish(satellaPeerGroupAdv,
                                                  DiscoveryService.GROUP,
                                                  PeerGroup.DEFAULT_LIFETIME,
                                                  PeerGroup.DEFAULT_EXPIRATION);
            rootPeerGroupDiscoveryService.remotePublish(satellaPeerGroupAdv,
                                                        DiscoveryService.GROUP,
                                                        PeerGroup.DEFAULT_EXPIRATION);
        } catch (java.io.IOException e) {
            System.err.println("Can't Publish satellaPeerGroupAdv");
            System.exit(1);
        }
        // Finally Create the Peer Group
        if (satellaPeerGroupAdv==null)
            System.err.println("satellaPeerGroupAdv is null");
        try{
            satellaPeerGroup=rootPeerGroup.newGroup(satellaPeerGroupAdv);
        } catch (net.jxta.exception.PeerGroupException e) {
            System.err.println("Can't create Satella Peer Group from Advertisement");
            e.printStackTrace();
            return null;
        }
        return satellaPeerGroup;
    }

    private PeerGroupAdvertisement createPeerGroupAdvertisement(
        ModuleImplAdvertisement passwdMembershipModuleImplAdv,
        String groupName, String login, String passwd) {
```

```

        // Create a PeerGroupAdvertisement for the peer group
        PeerGroupAdvertisement  satellaPeerGroupAdv=
           (PeerGroupAdvertisement) AdvertisementFactory.newAdvertisement(
           PeerGroupAdvertisement.getAdvertisementType());

        // Instead of creating a new group ID each time using the line below
        //    satellaPeerGroupAdv.setPeerGroupID(IDFactory.newPeerGroupID());
        // I use a fixed ID so that each time I start SecurePeerGroup,
        // it creates the same Group
        satellaPeerGroupAdv.setPeerGroupID(satellaPeerGroupID);
        satellaPeerGroupAdv.setModuleSpecID(
           passwdMembershipModuleImplAdv.getModuleSpecID());
        satellaPeerGroupAdv.setName(groupName);
       satellaPeerGroupAdv.setDescription("Peer Group using Password Authentication");

        // Now create the Structured Document containing the login and passwd
        // Login and passwd are put into the Param section of the peer group
        if (login!=null) {
            StructuredTextDocument loginAndPasswd =
               (StructuredTextDocument)
               StructuredDocumentFactory.newStructuredDocument(
               new MimeMediaType("text/xml"),"Parm");
            String loginAndPasswdString =
               login + ":" + PasswdMembershipService.makePsswd(passwd)+ ":";
            TextElement loginElement =
                loginAndPasswd.createElement("login", loginAndPasswdString);
            loginAndPasswd.appendChild(loginElement);
            // All right, now that loginAndPasswdElement (The strucuted document
            // that is the Param Element for The PeerGroup Adv)
            // is done, include it in the Peer Group Advertisement
            satellaPeerGroupAdv.putServiceParam(
               PeerGroup.membershipClassID,loginAndPasswd);
        }
        return satellaPeerGroupAdv;
    }

    private ModuleImplAdvertisement
     createPasswdMembershipPeerGroupModuleImplAdv(PeerGroup rootPeerGroup) {
        // Create a ModuleImpl Advertisement for the Passwd Membership Service
        // Take an allPurposePeerGroupImplAdv ModuleImplAdvertisement parameter and
        // clone some of its fields; this is easier than recreating everything
        // from scratch

        // Try to locate where the PasswdMembership is within this
       // ModuleImplAdvertisement.
        // For a PeerGroup Module Impl, the list of the services
        // (including Membership) are located in the Param section
        ModuleImplAdvertisement allPurposePeerGroupImplAdv=null;
        try {
            allPurposePeerGroupImplAdv=
               rootPeerGroup.getAllPurposePeerGroupImplAdvertisement();
        } catch (java.lang.Exception e) {
            System.err.println("Can't Execute 
allPurposePeerGroupImplAdv=rootPeerGroup.getAllPurposePeerGroupImplAdvertisement();");
            System.exit(1);
        }
        ModuleImplAdvertisement passwdMembershipPeerGroupModuleImplAdv =
            allPurposePeerGroupImplAdv;
        ModuleImplAdvertisement passwdMembershipServiceModuleImplAdv=null;
        StdPeerGroupParamAdv passwdMembershipPeerGroupParamAdv=null;

        try {
            passwdMembershipPeerGroupParamAdv = new
                StdPeerGroupParamAdv(allPurposePeerGroupImplAdv.getParam());
        } catch (net.jxta.exception.PeerGroupException e) {
            System.err.println("Can't execute: StdPeerGroupParamAdv 
passwdMembershipPeerGroupParamAdv = new StdPeerGroupParamAdv
(allPurposePeerGroupImplAdv.getParam());");
            System.exit(1);
        }

        Hashtable allPurposePeerGroupServicesHashtable =
            passwdMembershipPeerGroupParamAdv.getServices();
        Enumeration allPurposePeerGroupServicesEnumeration =
            allPurposePeerGroupServicesHashtable.keys();
        boolean membershipServiceFound=false;
        while ((!membershipServiceFound) &&
             (allPurposePeerGroupServicesEnumeration.hasMoreElements())) {
            Object allPurposePeerGroupServiceID =
                allPurposePeerGroupServicesEnumeration.nextElement();
            if (allPurposePeerGroupServiceID.equals(PeerGroup.membershipClassID)) {
                // allPurposePeerGroupMemershipServiceModuleImplAdv is the
                // all Purpose Mermbership Service for the all purpose
                // Peer Group  Module Impl adv
             ModuleImplAdvertisement allPurposePeerGroupMemershipServiceModuleImplAdv =
                   (ModuleImplAdvertisement)
                   allPurposePeerGroupServicesHashtable.get(allPurposePeerGroupServiceID);
                //Create the passwdMembershipServiceModuleImplAdv
                passwdMembershipServiceModuleImplAdv =
                   this.createPasswdMembershipServiceModuleImplAdv(
                   allPurposePeerGroupMemershipServiceModuleImplAdv);
                //Remove the All purpose Membership Service implementation
            allPurposePeerGroupServicesHashtable.remove(allPurposePeerGroupServiceID);
                // And Replace it by the Passwd Membership Service Implementation
                allPurposePeerGroupServicesHashtable.put(
                   PeerGroup.membershipClassID,passwdMembershipServiceModuleImplAdv);
                membershipServiceFound=true;
                // Now the Service Advertisements are complete
                // Let's update the passwdMembershipPeerGroupModuleImplAdv by
                // Updating its param
                passwdMembershipPeerGroupModuleImplAdv.setParam(
                   (Element) passwdMembershipPeerGroupParamAdv.getDocument(
                   new MimeMediaType("text/xml")));
                // Update its Spec ID
                // This comes from the Instant P2P PeerGroupManager Code
                if
(!passwdMembershipPeerGroupModuleImplAdv.getModuleSpecID().equals(PeerGroup.allPurposePeer
GroupSpecID)) {
                    passwdMembershipPeerGroupModuleImplAdv.setModuleSpecID(
IDFactory.newModuleSpecID(passwdMembershipPeerGroupModuleImplAdv.getModuleSpecID().getBase
Class()));
                }
                else {
                    ID passwdGrpModSpecID= ID.nullID;
                    try {
                        passwdGrpModSpecID = IDFactory.fromURL(new URL("urn","",
                       "jxta:uuid-"+"DeadBeefDeafBabaFeedBabe00000001" +"04" +"06" ) );
                    }
                    catch (java.net.MalformedURLException e) {}
                    catch (java.net.UnknownServiceException ee) {}
                    passwdMembershipPeerGroupModuleImplAdv.setModuleSpecID(
                       (ModuleSpecID) passwdGrpModSpecID);
                } //end else
                membershipServiceFound=true;
          } //end if (allPurposePeerGroupServiceID.equals(PeerGroup.membershipClassID))
        }//end while
        return passwdMembershipPeerGroupModuleImplAdv;
    }

        private  ModuleImplAdvertisement createPasswdMembershipServiceModuleImplAdv(
            ModuleImplAdvertisement allPurposePeerGroupMemershipServiceModuleImplAdv) {

        //Create a new ModuleImplAdvertisement for the Membership Service
        ModuleImplAdvertisement passwdMembershipServiceModuleImplAdv =
             (ModuleImplAdvertisement) AdvertisementFactory.newAdvertisement(
              ModuleImplAdvertisement.getAdvertisementType());

        passwdMembershipServiceModuleImplAdv.setModuleSpecID(
           PasswdMembershipService.passwordMembershipSpecID);
        passwdMembershipServiceModuleImplAdv.setCode(
           PasswdMembershipService.class.getName());
        passwdMembershipServiceModuleImplAdv.setDescription(
           " Module Impl Advertisement for the PasswdMembership Service");
        passwdMembershipServiceModuleImplAdv.setCompat(
           allPurposePeerGroupMemershipServiceModuleImplAdv.getCompat());
        passwdMembershipServiceModuleImplAdv.setUri(
           allPurposePeerGroupMemershipServiceModuleImplAdv.getUri());
        passwdMembershipServiceModuleImplAdv.setProvider(
           allPurposePeerGroupMemershipServiceModuleImplAdv.getProvider());
        return passwdMembershipServiceModuleImplAdv;
    }

    private PeerGroup discoverPeerGroup(PeerGroup myNetPeerGroup,
        PeerGroupID satellaPeerGroupID) {
        // First discover the peer group
        // In most cases we should use discovery listeners so that
        // we can do the discovery assynchroneously.
        // Here I won't, for increased simplicity and because
        // The Peer Group Advertisement is in the local cache for sure
        PeerGroup satellaPeerGroup;
        DiscoveryService myNetPeerGroupDiscoveryService=null;
        if (myNetPeerGroup!=null) {
            myNetPeerGroupDiscoveryService =
                myNetPeerGroup.getDiscoveryService();
        }
        else {
            System.err.println("Can't join Peer Group since its parent is null");
            System.exit(1);
        }
        boolean isGroupFound=false;
        Enumeration localPeerGroupAdvertisementEnumeration=null;
        PeerGroupAdvertisement satellaPeerGroupAdv=null;
        while(!isGroupFound) {
            try{
                localPeerGroupAdvertisementEnumeration=
                   myNetPeerGroupDiscoveryService.getLocalAdvertisements(
                   DiscoveryService.GROUP,"GID",satellaPeerGroupID.toString());
            }
            catch (java.io.IOException e) {
                System.out.println("Can't Discover Local Adv");
            }
            if (localPeerGroupAdvertisementEnumeration!=null) {
                while (localPeerGroupAdvertisementEnumeration.hasMoreElements()) {
                    PeerGroupAdvertisement pgAdv=null;
                    pgAdv=(PeerGroupAdvertisement)
                        localPeerGroupAdvertisementEnumeration.nextElement();
                    if (pgAdv.getPeerGroupID().equals(satellaPeerGroupID)) {
                        satellaPeerGroupAdv=pgAdv;
                        isGroupFound=true ;
                        break ;
                    }
                }
            }
            try {
               Thread.sleep(5 * 1000);
           } catch(Exception e) {}
        }
        try {
            satellaPeerGroup=myNetPeerGroup.newGroup(satellaPeerGroupAdv);
        } catch (net.jxta.exception.PeerGroupException e) {
            System.err.println("Can't create Peer Group from Advertisement");
            e.printStackTrace();
            return null;
        }
        return satellaPeerGroup;
    }

    private void joinPeerGroup(PeerGroup satellaPeerGroup,
                               String login,String passwd) {
        // Get the Heavy Weight Paper for the resume
        // Alias define the type of credential to be provided
        StructuredDocument creds = null;
        try {
            // Create the resume to apply for the Job
            // Alias generate the credentials for the Peer Group
            AuthenticationCredential authCred =
               new AuthenticationCredential( satellaPeerGroup, null, creds );

            // Create the resume to apply for the Job
            // Alias generate the credentials for the Peer Group
            MembershipService membershipService =
                satellaPeerGroup.getMembershipService();

            // Send the resume and get the  Job application form
            // Alias get the Authenticator from the Authentication creds
            Authenticator auth = membershipService.apply( authCred );

            // Fill in the Job Application Form
            // Alias complete the authentication
            completeAuth( auth, login, passwd );

            // Check if I got the Job
            // Alias Check if the authentication that was submitted was
            //accepted.
            if( !auth.isReadyForJoin() ) {
                System.out.println( "Failure in authentication.");
              System.out.println( "Group was not joined. Does not know how to complete
authenticator");
            }
            // I got the Job, Join the company
            // Alias I the authentication I completed was accepted,
            // therefore join the Peer Group accepted.
            membershipService.join( auth );
        } catch (Exception e){
            System.out.println("Failure in authentication.");
            System.out.println("Group was not joined. Login was incorrect.");
            e.printStackTrace();
        }
    }

    private  void completeAuth(Authenticator auth, String login,
                                String passwd) throws Exception{

        Method [] methods = auth.getClass().getMethods();
        Vector authMethods = new Vector();

        // Find out which fields of the application need to be filled
        // Alias Go through the methods of the Authenticator class and copy
        // them sorted by name into a vector.
        for( int eachMethod = 0; eachMethod < methods.length; eachMethod++ ) {
            if( methods[eachMethod].getName().startsWith("setAuth") ){
                if( Modifier.isPublic( methods[eachMethod].getModifiers())){

                    // sorted insertion.
                    for( int doInsert = 0; doInsert <= authMethods.size();
                    doInsert++ ) {
                            int insertHere = -1;
                            if( doInsert == authMethods.size() )
                                insertHere = doInsert;
                            else {

if(methods[eachMethod].getName().compareTo(((Method)authMethods.elementAt( doInsert
)).getName()) <= 0 )
                                    insertHere = doInsert;
                            } // end else

                            if(-1!= insertHere ) {
                                authMethods.insertElementAt(
                                methods[eachMethod],insertHere);
                                break;
                            } // end if ( -1 != insertHere)
                        } // end for (int doInsert=0
                    } // end if (modifier.isPublic
                } // end if (methods[eachMethod]
            } // end for (int eachMethod)

            Object [] AuthId = {login};
            Object [] AuthPasswd = {passwd};

            for( int eachAuthMethod=0;eachAuthMethod<authMethods.size();
            eachAuthMethod++ ) {
                Method doingMethod = (Method) authMethods.elementAt(eachAuthMethod);

                String authStepName = doingMethod.getName().substring(7);
                if (doingMethod.getName().equals("setAuth1Identity")) {
                    // Found identity Method, providing identity
                    doingMethod.invoke( auth, AuthId);

                }
                else if (doingMethod.getName().equals("setAuth2_Password")){
                    // Found Passwd Method, providing passwd
                    doingMethod.invoke( auth, AuthPasswd );
                }
            }
        }

        private void printXmlAdvertisement( String title, Advertisement adv) {
            // First, Let's print a "nice" Title
            String separator="";
            for (int i=0 ; i<title.length()+4; i++) {
                separator=separator+"-";
            }
            System.out.println(separator);
            System.out.println("| "+title+" |");
            System.out.println(separator);

            // Now let's print the Advertisement
            StringWriter outWriter = new StringWriter();
            StructuredTextDocument docAdv =
               (StructuredTextDocument)adv.getDocument(new MimeMediaType(
                   "text/xml"));
            try {
                docAdv.sendToWriter(outWriter);
            }
            catch (java.io.IOException e) {
                System.err.println("Can't Execute: docAdv.sendToWriter(outWriter);");
            }
            System.out.println(outWriter.toString());

            // Let's end up with a line
            System.out.println(separator);
        }

        /** Starts the jxta platform */
        private PeerGroup startJxta() {
            PeerGroup myNetPeerGroup  = null;
            try {
                myNetPeerGroup=PeerGroupFactory.newNetPeerGroup();
            }
            catch ( PeerGroupException e) {
                // could not instanciate the group, print the stack and exit
                System.out.println("fatal error : group creation failure");
                e.printStackTrace();
                System.exit(1);
            }
            return myNetPeerGroup;
        }

        public static void main(String args[]) {
            SecurePeerGroup satellaRoot = new SecurePeerGroup();
            System.exit(0);
        }
    }
```

CHAPTER 8

References

The following Web pages contain information on Project JXTA:

- *http://www.jxta.org* — home Web page for Project JXTA
- *http://spec.jxta.org* — Project JXTA specification
- *http://platform.jxta.org* — Project JXTA platform infrastructure and protocols for the J2SE platform binding
- *http://platform.jxta.org/java/api/overview-tree.html* — public API (Javadoc software)
- *http://www.jxta.org/Tutorials.html* — numerous Java tutorials

There are numerous technical white papers posted on *http://www.jxta.org/white_papers.html*. Those of particular interest to developers include:

- *Project JXTA: An Open, Innovative Collaboration*, Sun Microsystems white paper.
- *Project JXTA: A Technology Overview*, Li Gong, Sun Microsystems white paper.
- *Project JXTA Technology: Creating Connected Communities*, Sun Microsystems white paper.
- *Project JXTA Virtual Network*, Bernard Traversat et al., Sun Microsystems white paper.
- *Project JXTA: A Loosely-Consistent DHT Rendezvous Walker*, Bernard Traversat, Mohamed Abdelaziz, and Eric Pouyoul, Sun Microsystems white paper.
- *Introduction to the JXTA Abstraction Layer*, Neelakanth Nadgir and Jerome Verbeke, Sun Microsystems.
- *PKI Security for JXTA Overlay Networks*, Jeffrey Eric Altman, IAM Consulting.

APPENDIX **A**

Glossary

Advertisement	Project JXTA's language-neutral metadata structures that describe peer resources such as peers, peer groups, pipes, and services. Advertisements are represented as XML documents.
ASN.1	Abstract Syntax Notation One; a formal language for abstractly describing messages sent over a network. (See *http://www.asn1.org/* for more information.)
Binding	An implementation of the Project JXTA protocols for a particular environment (e.g., the J2SE platform binding).
Credential	A token used to uniquely identify the sender of a message; can be used to provide message authorization.
Endpoint	See *Peer Endpoint* and *Pipe Endpoint*.
ERP	Endpoint Routing Protocol; used by peers to find routes to other peers.
Gateway	See *Relay Peer*.
Input Pipe	A pipe endpoint; the receiving end of a pipe. Pipe endpoints are dynamically bound to peer endpoints at runtime.
J2SE	Java 2 Platform, Standard Edition software.
Message	The basic unit of data exchange between peers; each message contains an ordered sequence of named sub-sections, called message elements, which can hold any form of data. Messages are exchanged by the Pipe Service and the Endpoint Service.
Message Element	A named and typed component of a message (i.e., a name/value pair).
Module	An abstraction used to represent any piece of "code" used to implement a behavior in the JXTA world. Network services are the mode common example of behavior that can be instantiated on a peer.
Module Class	Represents an expected behavior and an expected binding to support the module; is used primarily to advertise the existence of a behavior.

Module Implementation	The implementation of a given module specification; there may be multiple module implementations for a given module specification.
Module Specification	Describes a specification of a given module class; it is one approach to providing the functionality that a module class implies. There can be multiple module specifications for a given module class. The module specification is primarily used to access a module.
NAT	Network Address Translation. Network Address Translation allows a single device, such as a router, to act as an agent between the Internet (or "public network") and a local (or "private") network.
Output Pipe	A pipe endpoint; the sending end of a pipe. Pipe endpoints are dynamically bound to peer endpoints at runtime.
P2P	Peer-to-peer; a decentralized networking paradigm in which distributed nodes, or peers, communicate and work collaboratively to provide services.
PBP	Peer Binding Protocol; used by peers to establish a virtual communication channel, or pipe, between one or more peers.
PDP	Peer Discovery Protocol; used by peers to discover resources from other peers.
Peer	Any networked device that implements one or more of the JXTA protocols.
Peer Endpoint	A URI that uniquely identifies a peer network interface (e.g., a TCP port and associated IP address).
Peer Group	A collection of peers that have a common set of interests and have agreed upon a common set of services.
Peer Group ID	ID that uniquely identifies a peer group.
Peer ID	ID that uniquely identifies a peer.
PIP	Peer Information Protocol; used by peers to obtain status information (uptime, state, recent traffic, etc.) from other peers.
Pipe	An asynchronous and unidirectional message transfer mechanism used by peers to send and receive messages; pipes are bound to specific peer endpoints, such as a TCP port and associated IP address.
Pipe Endpoint	Pipe endpoints are referred to as *input pipes* and *output pipes*; they are bound to peer endpoints at runtime.
PKI	Public Key Infrastructure. Supports digital signatures and other public key-enabled security services.
PRP	Peer Resolver Protocol; used by peers to send generic queries to other peer services and receive replies.

Relay Peer	Maintains information on routes to other peers, and helps relay messages to peers. (Previously referred to as router peer.)
Rendezvous Peer	Maintains a cache of advertisements and forwards discovery requests to other rendezvous peers to help peers discover resources.
RVP	Rendezvous Protocol; responsible for propagating messages within a peer group.
TLS	Transport Layer Security. (See *http://www.ietf.org/html.charters/tls-charter.html* for more details.)
URI	Uniform Resource Identifier. A compact string of characters for identifying an abstract or physical resource. (See *http://www.w3.org/Addressing/URL/URI_Overview.html* for more details.)
URN	Uniform Resource Name. A kind of URI that provides persistent identifiers for information resources. (See IETF RFC 2141, *http://www.ietf.org/rfc/rfc2141.txt*, for more details.)

APPENDIX **B**

Troubleshooting

This appendix discusses commonly encountered problems compiling and running JXTA applications.

Errors compiling JXTA applications

Check that your are including the correct `jxta.jar` file in your compilation statement (`-classpath` option). If you have downloaded multiple versions, verify that you are including the most recent version in your compilation statement.

Note – The required `.jar` files can be downloaded from the JXTA Web site: *http://download.jxta.org*.

Errors running JXTA applications

Setting the classpath variable

When you run your JXTA application, you need to set the `-classpath` variable to indicate the location of the required `.jar` files. Be sure to include the same version that you used when compiling your JXTA application. Although you need only the `jxta.jar` file for compilation, you need multiple `.jar` files when running a JXTA application.

Note – See Table 6-1 on page 24 for a list of the required Java `.jar` files.

Unable to discover JXTA peers

If you are unable to discover other JXTA resources (peers, peer groups, or other advertisements), you may have configured your JXTA environment incorrectly. Common configuration issues include the following:

- If you are located behind a firewall or NAT, you must use HTTP and specify a relay node.
- If you are using TCP with NAT, you may need to specify your NAT public address.
- You may need to specify at least one rendezvous node.

Remove the JXTA configuration file (`PlatformConfig`) and then re-run your application. When the JXTA Configurator window appears, enter your configuration information. See Appendix B, *Configuration Tool* for more details on running the JXTA Configurator.

Using the JXTA Shell

You can use the JXTA Shell to help troubleshoot configuration issues and test JXTA services. Commands are available to discover JXTA advertisements, create JXTA resources (e.g., groups, pipes, messages, and advertisements), join and leave peer groups, send and receive messages on a pipe, and much more.

For example, to verify correct network configuration you can use the JXTA Shell command "rdvstatus" to display information about your current rendezvous status (i.e., if you are configured as a rendezvous peer, and who your current rendezvous peers are). You can also use "search -r" to send out discovery requests, and then use "peers" to display any peers that have been discovered — to confirm that network connectivity is working as expected.

For more information on downloading and using the JXTA Shell, please see:

http://shell.jxta.org/

Starting from a clean state

Some problems can be caused by stale configuration or cache information. Try removing the JXTA configuration files and cache directory:

- `./.jxta/PlatformConfig`
- `./.jxta/cm` (directory)

Re-launch the application. When the Configuration window appears, enter the appropriate information for your network configuration. See *http://platform.jxta.org/java/confighelp.html* for more details on running the JXTA Configurator.

Displaying additional log information

If your JXTA application isn't behaving as you expect, you can turn on additional logging so that more information is displayed when your application runs.

To select a new logging, or trace, level, re-run the JXTA Configurator and from the Advanced Settings tab select the desired Trace Level from the pull-down menu. The default trace level is *error; warn*, *info*, and *debug* levels provide more information. For more information on running the JXTA Configurator, please see Appendix B, *Configuration Tool* on page 123.

You can also choose to edit the `PlatformConfig` file in the current directory rather than re-running the JXTA Configurator. For example, the following entry in `PlatformConfig` sets the trace level to "warning":

```
<Dbg>
   warn
<\Dbg>
```

Removing User name or Password

The first time you run a JXTA application, you will be prompted to enter a user name and password. Each subsequent time you run the application, you will be prompted to enter the same user name and password pair. If you forget either the user name or the password, you can remove the `pse` directory (located in the current directory) and then re-run the application. The JXTA Configurator will be displayed, and you can enter a new user name and password. See *http://platform.jxta.org/java/confighelp.html* for more details on running the JXTA Configurator.

All-Purpose Peer Group Module Implementation Advertisement

This appendix contains an example XML advertisement of an all-purpose peer group module implementation. It is used in the programming example which creates a secure peer group (see “Creating a Secure Peer Group” on page 102).

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

<!DOCTYPE jxta:MIA>

<jxta:MIA xmlns:jxta="http://jxta.org">
   <MSID>
      urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000010306
   </MSID>
   <Comp>
      <Efmt>
         JDK1.4
      </Efmt>
      <Bind>
         V1.0 Ref Impl
      </Bind>
   </Comp>
   <Code>
      net.jxta.impl.peergroup.StdPeerGroup
   </Code>
   <PURI>
      http://www.jxta.org/download/jxta.jar
   </PURI>
   <Prov>
      sun.com
   </Prov>
   <Desc>
      General Purpose Peer Group Implementation
   </Desc>
```

```
<Parm>
   <Svc>
      <jxta:MIA>
         <MSID>
            urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000060106
         </MSID>
         <Comp>
            <Efmt>
               JDK1.4
            </Efmt>
            <Bind>
               V1.0 Ref Impl
            </Bind>
         </Comp>
         <Code>
            net.jxta.impl.rendezvous.RendezVousServiceImpl
         </Code>
         <PURI>
            http://www.jxta.org/download/jxta.jar
         </PURI>
         <Prov>
            sun.com
         </Prov>
         <Desc>
            Reference Implementation of the Rendezvous service
         </Desc>
      </jxta:MIA>
   </Svc>

   <Svc>
      <jxta:MIA>
         <MSID>
            urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000030106
         </MSID>
         <Comp>
            <Efmt>
               JDK1.4
            </Efmt>
            <Bind>
               V1.0 Ref Impl
            </Bind>
         </Comp>
         <Code>
            net.jxta.impl.discovery.DiscoveryServiceImpl
         </Code>
         <PURI>
            http://www.jxta.org/download/jxta.jar
         </PURI>
         <Prov>
```

```
          sun.com
       </Prov>
       <Desc>
          Reference Implementation of the DiscoveryService service
       </Desc>
   </jxta:MIA>
</Svc>

<Svc>
   <jxta:MIA>
       <MSID>
          urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000050106
       </MSID>
       <Comp>
          <Efmt>
             JDK1.4
          </Efmt>
          <Bind>
             V1.0 Ref Impl
          </Bind>
       </Comp>
       <Code>
          net.jxta.impl.membership.NullMembershipService
       </Code>
       <PURI>
          http://www.jxta.org/download/jxta.jar
       </PURI>
       <Prov>
          sun.com
       </Prov>
       <Desc>
          Reference Implementation of the MembershipService service
       </Desc>
   </jxta:MIA>
</Svc>

<Svc>
   <jxta:MIA>
       <MSID>
             urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000070106
       </MSID>
       <Comp>
          <Efmt>
             JDK1.4
          </Efmt>
          <Bind>
             V1.0 Ref Impl
          </Bind>
       </Comp>
```

```
        <Code>
            net.jxta.impl.peer.PeerInfoServiceImpl
        </Code>
        <PURI>
            http://www.jxta.org/download/jxta.jar
        </PURI>
        <Prov>
            sun.com
        </Prov>
        <Desc>
            Reference Implementation of the Peerinfo service
        </Desc>
    </jxta:MIA>
</Svc>

<Svc>
    <jxta:MIA>
        <MSID>
            urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000020106
        </MSID>
        <Comp>
            <Efmt>
                JDK1.4
            </Efmt>
            <Bind>
                V1.0 Ref Impl
            </Bind>
        </Comp>
        <Code>
            net.jxta.impl.resolver.ResolverServiceImpl
        </Code>
        <PURI>
            http://www.jxta.org/download/jxta.jar
        </PURI>
        <Prov>
            sun.com
        </Prov>
        <Desc>
            Reference Implementation of the ResolverService service
        </Desc>
    </jxta:MIA>
</Svc>

<Svc>
    <jxta:MIA>
        <MSID>
            urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE000000040106
        </MSID>
        <Comp>
```

```
            <Efmt>
               JDK1.4
            </Efmt>
            <Bind>
               V1.0 Ref Impl
            </Bind>
         </Comp>
         <Code>
            net.jxta.impl.pipe.PipeServiceImpl
         </Code>
         <PURI>
            http://www.jxta.org/download/jxta.jar
         </PURI>
         <Prov>
            sun.com
         </Prov>
         <Desc>
            Reference Implementation of the PipeService service
         </Desc>
      </jxta:MIA>
   </Svc>

   <App>
      <jxta:MIA>
         <MSID>
            urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000C0206
         </MSID>
         <Comp>
            <Efmt>
               JDK1.4
            </Efmt>
            <Bind>
               V1.0 Ref Impl
            </Bind>
         </Comp>
         <Code>
            net.jxta.impl.shell.bin.Shell.Shell
         </Code>
         <PURI>
            http://www.jxta.org/download/jxta.jar
         </PURI>
         <Prov>
            sun.com
         </Prov>
         <Desc>
            JXTA Shell reference implementation
         </Desc>
      </jxta:MIA>
   </App>
   </Parm>
</jxta:MIA>
```