JXTA Java™ Standard Edition v2.5: Programmers Guide

June 7, 2007

Please Recycle

Table of Contents

Introduction....7
Why JXTA™ ?....7
What is JXTA™ Technology?....8
What can be done with JXTA™ Technology? 8
Where to get JXTA™ technology....9
Getting Involved....9
JXTA™ Architecture....10
Overview....10
JXTA Components....11
Key Aspects of the JXTA Architecture....11
JXTA Concepts....12
Peers....12
Peer Groups....12
Network Services....13
Peer Group Services....13
Modules....14
Messages....15
Design points for JXTA Messages....16
Design points for Message Elements....17
Pipes....17
JxtaSocket and JxtaBiDiPipe....18
Advertisements....19
Security....21
IDs....21
Network Architecture....22
Network Organization....22
Shared Resource Distributed Index (SRDI)....23
Queries....23
Firewalls and NAT....26
JXTA Protocols....28
Peer Discovery Protocol....29
Peer Information Protocol....29
Peer Resolver Protocol....29
Pipe Binding Protocol....30
Endpoint Routing Protocol....30
Rendezvous Protocol....31
The Basics....32
Getting Started....32
Accessing On-line Documentation 32
Downloading Binaries....32
Compiling JXTA Code....33
Running JXTA Applications....33

"Public JXTA Network" Configuration Setting....34
JXTA and HTTP Proxies....34
Do I need a Proxy?....34
Creating various ID types....35
ID Tutorial source code....36
Advertisements....39
Advertisement Tutorial source....40
Messages and Message Elements....49
Message and Message Element tutorial source....50
Programming with JXTA....57
HelloWorld Example....57
Running the HelloWorld Example....57
Source Code: HelloWorld....58
Peer Discovery....59
Discovery Service....59
DiscoveryClient....60
Source Code: DiscoveryClient....63
Source Code: DiscoveryServer....66
Membership Service....69
Role of the Membership Service....69
Membership Service and Credentials....69
Local Credentials and Remote Credentials....69
Membership Services and Peer Groups....69
Membership Service and Other Services....69
Membership Service Basic Operation....70
Remote Credentials & Access Control....73
PSE Membership Service....75
Introduction....75
The PSE Membership Service and Other Services....75
The PSE Membership Service and Keystores....75
Accessing the PSE Membership Service....75
PSE Membership Service Components....75
PSE Membership First Steps....76
The PSE Credential....76
PSE Config....77
Pipe Service....79
JXTA Pipe Service....79
PipeServer....80
Source Code: PipeClient....82
Source Code: PipeServer....85
Multicast Socket....89
Description....89
Learning goals....89
Basic Operations....89
Caveats....89
Source Code : JxtaMulticastSocketClient....90

Source Code : JxtaMulticastSocketServer....92
Propagated Pipe Example....94
PropagatedPipeClient....94
Source Code: PropagatedPipeClient....96
Source Code : PropagatedPipeServer....99
JxtaBiDiPipe Example....104
JxtaBiDiPipe....104
JxtaServerPipeExample....105
Source Code: JxtaServerPipeExample....106
JxtaBidiPipeExample....111
Source Code: JxtaBidiPipeExample....112
JxtaSocket Tutorial....115
JxtaServerSocketExample....116
Source Code: SocketServer....117
Pipe adertisement example....120
ClientSocket....121
Source Code: SocketClient....122
JXTA Services....125
Creating a JXTA Service....126
Server....128
readMessages()....129
Source Code: Service Server....131
Example Service Advertisement:....135
Service Client....136
Source Code: Service Client....138
Password Protected Peer Group....141
The constructor method PrivatePeerGroup()....142
createPeerGroup()....142
createPasswdMembershipPeerGroupModuleImplAdv()....143
createPeerGroupAdvertisement()....145
discoverPeerGroup()....145
joinPeerGroup()....145
completeAuth()....146
Source Code: PrivatePeerGroup....147
Glossary....156
Appendix I: References....160
Appendix II: Troubleshooting....161
Errors compiling JXTA applications....161
Errors running JXTA applications....161
Setting the classpath variable....161
Unable to discover JXTA peers....161
Using the JXTA Shell....161
Starting from a clean state....161
Starting from a clean state....161
Displaying additional log information....162
Removing User name or Password....162

Chapter 1:Introduction

JXTA™ is a set of open, generalized peer-to-peer (P2P) protocols that allow any networked device — sensors, cell phones, PDAs, laptops, workstations, servers and supercomputers — to communicate and collaborate mutually as peers. The JXTA protocols are programming language independent, and multiple implementations, also known as bindings, exist for different environments. Their common use of the JXTA protocols means that they are all fully interoperable. This guide focuses upon the JXTA binding for the Java™ Platform, Standard Edition software (JSE™). Similar programmers guide documents are being prepared for the Java™ Platform Micro Edition (J2ME™) and C/C++/.NET JXTA bindings.

The target audience for this guide is 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 JXTA technology, describes the JXTA network architecture and key concepts, and includes examples and discussion of essential programming constructs using the JXTA platform JSE binding.

Why JXTA™?

As the Web continues to grow in both content and the number of connected devices, peer-to-peer computing is becoming increasingly prevalent. Popular examples include 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, P2P application development is inefficient, with developers solving the same problems and duplicating similar infrastructure implementations. And, most applications are specific to a single platform and are unable to communicate and share data with other applications.

A primary design principal of JXTA is to provide a platform that embodies the basic P2P network functions. As such, JXTA overcomes potential shortcomings of many existing P2P systems:

- *Interoperability* — JXTA technology is designed to enable peers provisioning P2P services to locate and communicate with one another independent of network addressing and physical protocols.
- *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, the edge often being occasionally connected devices that are assigned non static addresses (e.g. DHCP). Because they are subject to unpredictable connectivity with potentially variable network addresses, they are outside the standard scope of DNS. JXTA empowers peers on the edge of the network by provisioning a globally unique peer addressing scheme that is independent of traditional name services. Through the use of JXTA IDs, a peer can migrate across physical networks, changing transports and network addresses, even being temporarily disconnected, and still be addressable by other peers.

What is JXTA™ Technology?

JXTA is an open network computing platform designed for peer-to-peer (P2P) computing by way of providing the basic building blocks and services required to enable anything anywhere application connectivity.

The name "JXTA" is not an acronym. It is short hand 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 backed with open source reference implementations 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 resources
- Communicate with each other
- Monitor each other

The JXTA protocols are designed to be independent of programming languages and transport protocols alike. The protocols can be implemented in the Java programming language, C/C++, .NET, Perl, and numerous other languages. Furthermore, they can be implemented on top of TCP/IP, HTTP, Bluetooth, or other transport protocols all the while maintaining global interoperability.

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 and NATs
- Easily share resources with anyone across the network
- Find up to the available 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 JXTA™ technology

Information on the JXTA technology can be found at the JXTA Web site http://www.jxta.org. Resources include project information, documentation, mailing lists, source code, binaries, documentation, and tutorials.

Getting Involved

As with any open source project, a primary goal is to get the community involved by contributing to JXTA. Two suggestions for getting started include joining a 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* — JXTA announcements and general information
  - *dev@jxta.org* — technical issues for developers
  - *user@jxta.org* — issues for new JXTA developers and users
  - guide@jxta.org — technical issues regarding this guide for developers and users

- Chat with other JXTA enthusiasts
  You can learn about the JXTA technology using the Shell interactive application and chat with other JXTA users and contributors using the myJXTA application which can be downloaded at:
  http://download.jxta.org/demo.
  The demonstration applications are available for the following platforms: Microsoft Windows, Solaris™ Operating Environment, Linux, UNIX, Mac OS X, and other Java technology enabled platforms

As you gain experience working with the JXTA technology, you may contribute by filing bug reports, writing or extending tutorials, contributing to existing projects, and proposing new projects. The JXTA JSE Project Homepage (see:http://platform.jxta.org) contains additional instructions for contributing.

Chapter 2:JXTA™ Architecture

Overview

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

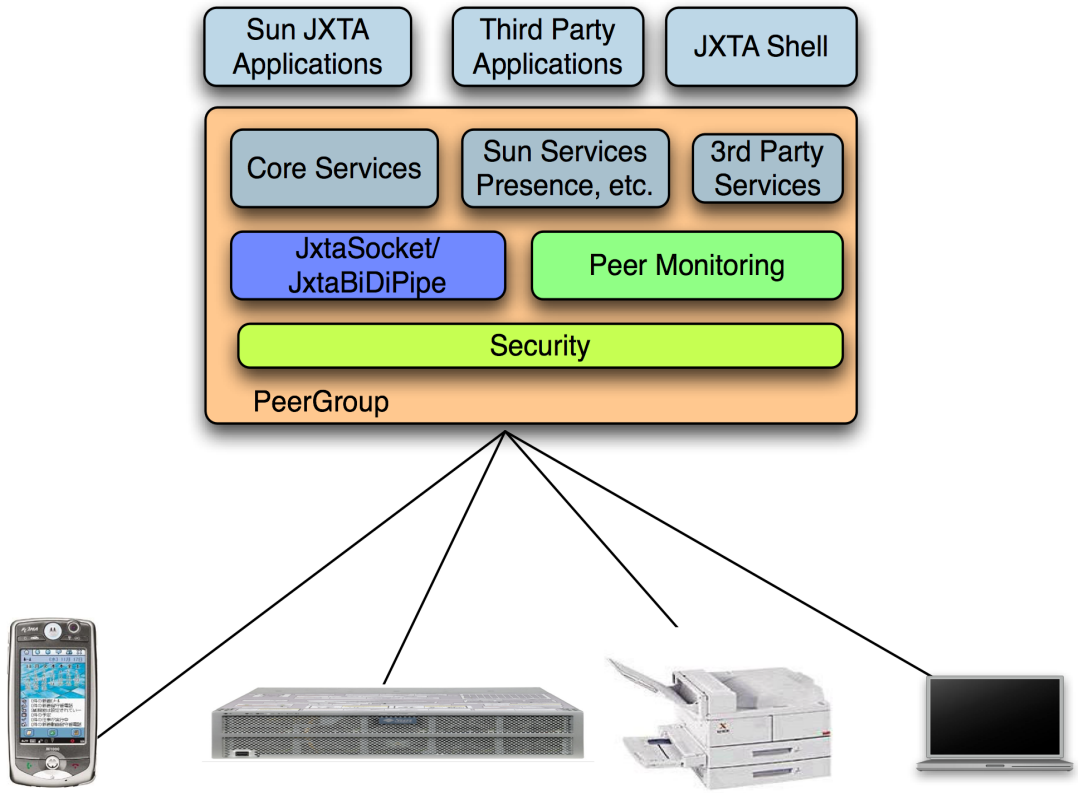


Figure 2: Jxta Software Architecture

- *Platform Layer (JXTA Core)*

  The platform layer, also known as the JXTA core, encapsulates the minimal and essential primitives that are common to P2P networking. It includes building blocks to enable key mechanisms for P2P applications, including discovery, communication transports (including firewall and NAT traversal), 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 a 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. One customer's application 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*. A peer may be any type of device from a sensor to a supercomputer or even a virtual process. Multiple peers may run on a single physical device and, potentially, multiple physical devices could cooperate to act as a single peer. The peers may be connected by any suitable networking protocol including TCP/IP, HTTP, Bluetooth, GSM, etc.

Each peer provides a set of *services* and *resources* which it makes available to other peers. Services are interactive programs and can include databases, authentication systems, chat servers or almost any program that can be networked. Two types of services are common within JXTA networks, *peer services* and *group services*. Peer services are those provided by a single peer. Group services are services which are provided in either a federated, redundant or cooperative way by the "whole group". Each Peer service instance is normally independent of other instances. Actions taken with one instance have no effect upon other instances. Each Peer group service instance is normally a participant in a common instance. Actions taken with one instance may (likely) have effects upon all instances.

All JXTA peers implement a small number of required *core services* and commonly also provide several additional *standard services*. Each Peer Group includes as part of it's definition the set of Group services which each peer must run in order to participate in the peer group.

A peer's resources are normally static (non-interactive) content which the peer either controls, owns or even merely has a copy of. Resources can include files, documents, media, advertisements, indexes but can also include real world resources such as switches, sensors and printers.

JXTA peers advertise their services and resources using XML documents called *advertisements.* Advertisements enable peers on the network to discover resources and services and to determine how to connect to and interact with those services.

Peers can organize themselves into *peer groups.* A Peer group, loosely defined, is any set of peers that provision and leverage a common set of services for a common purpose. There are two key aspects to this definition--common services and common purpose. Two peer groups might have the same set of services, for example a chat application, but different purposes, for example politics chat and sports chat. Peer groups can be defined on almost any basis that developers or deployers choose. For the preceding example the peer group could be redefined as providing a chat application for multiple topics but located within an organization, for example a university department. When defining a peer group the first two questions which must always be answered are; "What peers are members of this group?", and "What application or service are the peers cooperating to provide?".

JXTA peers use *sockets* and *pipes* to send *messages* to one another. JXTA sockets are reliable bi-directional connections used for applications to communicate reliably. 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 discussed in greater detail in the following chapters.

Key Aspects of the JXTA Architecture

Four essential aspects of the JXTA architecture that distinguish it from other distributed network models are:

- 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 (IDs).
- A decentralized search infrastructure based on Distributed Hash Table (DHT) for resource indexing.

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 entity that implements one or more of the JXTA protocols. Peers can reside on 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 addresses for use with the JXTA protocols. Each published address is advertised as a *peer endpoint,* which identifies the network address. Peer endpoints are used by peers to establish direct point-to-point connections between two peers.

Direct point-to-point network connections are not always available between peers. Intermediary peers may be used to route messages to peers that are separated due to physical network boundaries. The network boundaries can be natural boundaries such as between Ethernet and Bluetooth networks or artificially created due to network configuration. Artificial barriers can include NAT, firewalls and proxies. The use of enlisted intermediate peers can and will change over time with no impact on the JXTA application.

Peers are typically configured to spontaneously discover each other on the network to form relationships known as peer groups, which can be transient or persistent in nature.

JXTA peers can be categories into three main types:

- *Minimal-Edge peers*: Peers that implement only the required core JXTA services and may rely on other peers to act as their proxy for other services to fully participate in a JXTA Network. The proxy peers act as proxy for the non-core services. Typical minimal-edge peers include sensor devices and home automation devices,
- *Full-Edge Peer*: Peers that implements all of the core and standard JXTA services and can participate in all of the JXTA protocols. These peers form the majority of peers on a JXTA network and can include phones, PC's, servers, etc.
- Super-Peer: Peers that implement and provision resources to support the deployment and operation of a JXTA network. There are three key JXTA Super Peer functions. A single peer may implement one or more of these functions.
    - **Relay**: Used to store and forward messages between peers that do not have direct connectivity because of firewalls or NAT. Only peers which are unable to receive connections from other peers require a relay.
    - **Rendezvous**: Maintains global advertisement indexes and assists edge and proxied peers with advertisement searches. Also handles message broadcasting.
    - **Proxy**: Used by minimal-edge peers to get access to all the JXTA network functionalities. The proxy peer translates and summarizes requests, responds to queries and provides support functionality for minimal-edge peers.

These categories describe the most common peer confiugrations. Depending upon the application and peer capabilities it may make sense to deploy the peers with a mix of functionality. For example, it my be reasonable to deploy peers with full Discovery and Pipe functionality but require a proxy for running group services.

Peer Groups

A *peer group* is a collection of peers that have agreed upon a common set of services, or interests. Peers self-organize into peer groups, each of which is uniquely identified by a peer group ID. Each peer group establishes its own membership policy including open (anybody can join) to highly secure and protected (requiring credentials to gain membership).

Peers can belong to more than one peer group simultaneously. By default, the first group that is instantiated is the Network Peer Group. All peers belonging to the Network Peer Group and may choose to join additional peer groups at any time.

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. A group join is simply instantiating all the peer group services defined by the peer group. 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 user name/password exchange, or as sophisticated as public key cryptography. Peer group boundaries permit member peers to access and publish protected content. Peer groups form logical regions whose boundaries limit access to the peer group's 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 can 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.

Network Services

Peers cooperate and communicate to publish, discover, and invoke *network services*. Peers can publish multiple services which, in turn, are discovered 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 comprised of a collection of instances of the service, potentially cooperating with one another, running on multiple members of the hosting peer group. If any one peer fails, the collective peer group service is not affected as long as the service is still available from at least one peer member. Peer group services are published as part of the peer group advertisement.

Services can be either pre-installed into 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 analogous to performing a search on the Internet for a Web page, retrieving the page, and then a plug-in which is required by the page.

Peer Group Services

A peer group provides a set of services collectively known as peer group services. JXTA defines a core set of peer group services. These peer group services form the service signature of the peer group, as every peer that joins the group needs to implement these services. Additional services can be developed for delivering specific services. In order for two peers to interact via a service, they must both be members of the same peer group.

The core peer group services which every peer must implement in order to participate in the JXTA Network are:

- *Endpoint Service* — The endpoint service is used to send and receive Messages between peers. The Endpoint Service commonly implements the Endpoint Routing Protocol though minimal implementations provide only a small portion of the complete protocol.
- *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., find advertisements, determine

the status of a service or the availability of a pipe endpoint).

In addition to the core peer group services required of each peer several additional standard services are commonly defined for each peer group. Not all of the standard services must be implemented by every peer group. Depending upon the peer group definition these services may be required. A peer group specifies only the services it finds useful. A peer group may also rely on the services of the default net peer group to provide generic implementations of non-critical core services. The standard peer group services which are found in most peer groups are:

- *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 peer members to securely establish identities and trust within a peer group. Identities allow applications and services to determine who is requesting an operation and decide whether that operation should be allowed. Applications may perform their own access control or may use the Access Service. The Peer Group definition specifies the policies by which a peer may establish an identity.
- *Access Service* — The access service is used to validate requests made by one peer to another. The peer receiving the request checks the requesting peer's 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.
- *Monitoring Service* — The monitoring service is used to allow one peer to monitor other members of the same peer group.

Modules

JXTA modules are a low-level JXTA abstraction used to represent any piece of "code" and the interface (API) which that code provides. Modules are used to implement services, message transports and other loadable bits of JXTA code. Most JXTA developers typically don't have to deal with modules as the platform provides the initial set of services required by prototypical applications. The module abstraction does not specify the physical "code" implementation, as it can be provided as 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 implementer. 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 provide a generic abstraction to allow a peer to instantiate a function or service. When a peer joins a peer group they may find new behaviors that they may want to instantiate. For example, when joining a peer group, a peer may be required to provide 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 specified behavior.

The ability to describe and publish platform-independent behavior is essential to support the development of new peer group services which are provisioned by a heterogeneous cadre of peers. The module advertisement enables JXTA peers to describe a behavior in a platform-independent manner. In fact, the JXTA platform uses module advertisements to self-describe it's own services.

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 ModuleClassID (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 group 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.

Messages

JXTA services and applications communicate using JXTA Messages. JXTA Messages are the basic unit of data exchange between peers. Each JXTA protocol is defined as a set of messages which the participating peers exchange. Messages are sent between peers using the Endpoint Service and the Pipe Service as well as JxtaSocket and other approaches. Most applications do not need to use unidirectional pipe or the JXTA Endpoint Service directly. Instead, applications and services commonly use the JXTA Socket and JxtaBiDiPipe communication channels to send, and receive messages.

The JXTA protocols are specified as a set of messages exchanged between peers. The use of XML messages to define protocols allows many different kinds of peers to utilize a given 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 ignore the remainder. Each software platform binding describes how a message is converted to and from a native data structure such as a Java object or a C structure.

The JXTA protocols define two “on-wire” representations for messages: XML and binary. These on-wire representations are the data format used for transmitting the message between peers. Different on-wire formats are used to take best advantage of the characteristics of the underlying network transport. JXSE uses the binary message format.

A JXTA Message is a container for protocol information and content. Each message is composed of a series of Message Elements. The Message Elements contain the protocol information and content of the message. Each element contains a portion of the message. The purpose and content of each element is defined by the message protocol definition. Message Elements can contain any type of data required by the message protocol.

A JXTA Message consists of an ordered sequence of zero or more Message Elements. The same Message Element may appear multiple times within a Message. Each Message Element within a Message is associated with a namespace. Namespaces are provided to allow multiple protocol layers to use the same Message without their Message Elements colliding. For example, all of the core JXTA protocols store their protocol information into the reserved `'jxta'` namespace. The default namespace is reserved for the application or service's use. Applications may also define their own namespace for their Message Elements, but should avoid using the `'jxta'` namespace or any namespace defined by a module specification ID (MSID) other than their own MSID.

A JXTA Message Element may be used to store any type of data. Several varieties of Message Element are provided by the JXTA JSE API and additional Message Element types may be defined. The provided element classes allow creation of Message Elements from Strings, byte arrays, documents, input streams, etc. In order to be transmitted as part of a

Message the data of a element must be converted to raw bytes before it can be sent. The various “flavours” of message elements exist to provide convenient and efficient conversion to raw bytes. All Message Element classes merely convenient proxies for the binary form of the element data.

In developing programs using Message Elements you should avoid consider the source type of the Message Element after it's creation. ie. `MessageElement elem = new StringMessageElement(“foo”, “bar”, null);` rather than `StringMessageElement elem = new StringMessageElement(“foo”, “bar”, null);` Using instance variables of only type `MessageElement` reinforces the commonality of Each Message Element—all data is transmitted and received as bytes. Each Message Element is composed of four components:

1. An optional name which may be any string. Unnamed message elements are assumed to have the name "" (the empty string).
2. An optional MIME Media type. If no type is specified the Mime Media Type is assumed to be "Application/Octet-Stream", the default for binary data.
3. The message element content, a sequence of bytes. In order to be transmitted as part of a message the content of a message element must be converted to raw bytes before it can be sent. Message elements can be created from a variety of data types including Strings, byte arrays, documents, binary data, etc. In each case, regardless of the form of the source data, the resulting Message Element is a container for a sequence of bytes. All message elements are sent and received as raw bytes. The various “flavours” of message elements exist to provide convenient and efficient conversion to raw bytes.
4. An optional signature: Each message element can also be associated with an additional message element which may contain a cryptographic signature/hash of the message element.

Design points for JXTA Messages

- There is approximately 50 bytes of transmission overhead for each Message Element you add to a Message. If possible avoid protocol designs which use large numbers of small elements.
- Namespaces are only associated with Message Elements as the elements are added to a Message. Namespaces are dynamically associated with Message Elements in Messages because the appropriate namespace for an element might vary based upon context.
- There are four common ways that applications and services use message namespaces;
  - Use only the default namespace. Since this namespace is reserved for the application or service which creates the Message it is the easiest and most common choice.
  - Use a custom hardcoded namespace such as “myApplication”. This usage is common in JXTA applications and allows an application to define as many namespaces as needed. Though it is unlikely, there is a small risk of two separate usages of the same namespace colliding when using hardcoded namespace names. If convenient, applications and services should use their MSID as their namespace instead.
  - Use a custom namespace which is their MSID. Every MSID based namespace is reserved for use by applications or services which implement that module specification. For example, the MSID of the standard JXTA Rendezvous service is `urn:jxta:uuid-deadbeefdeafbabafeedbabe000000060106`. The namespace “`urn:jxta:uuid-deadbeefdeafbabafeedbabe000000060106`” may be used only by JXTA Rendezvous Service implementations which implement that specification.
  - Use a unique random namespace. The MessageElement class provides a utility for creating unique random names based upon UUIDs. These random names can be used by applications and services for single messages or sequences of messages. Applications can also generate their own random namespace names as long as the names are chosen to be unique. Most often this means using UUIDs or secure message digests (SHA1, MD5, etc.) as namespace names.
- Messages can be reused and are relatively cheap to clone. In some applications it may be convenient to build template messages which are then cloned and customized before sending rather than creating a completely new message each time.
- Messages are not synchronized and can only be used reliably from a single thread. This also applies while a message is being sent. Until sending is complete the message should not be used.

Design points for Message Elements

- Message Elements are immutable and can be re-used as part of multiple messages. This feature is commonly used for constant values such as addressing data or for cached data such as Peer Advertisements.
- The data contained within a MessageElement is accessible in four ways:
    1. As an `java.io.InputStream`
    2. Sending the data a `java.io.OutputStream`
    3. As a `java.lang.String`
    4. As a byte array

Pipes

JXTA peers use *pipes* to send messages to one another. Pipes are an asynchronous, unidirectional and non-reliable (with the exception of unicast secure pipes) message transfer mechanism used for communication and data transfer. Pipes are virtual communication channels and may connect peers that do not have a direct physical link, resulting in a logical connection. Pipes can be used to send any type of data including XML and HTML text, images, music, binary code, data strings and Java Objects.

The pipe endpoints are referred to as the receiving *input pipe* and the sending *output pipe*. Pipe endpoints are dynamically bound to peer endpoints when the pipe is opened. Peer endpoints correspond to available peer network interfaces with an example being a TCP port and associated IP address, that can be used to send and receive messages. JXTA pipes can have endpoints that are connected to different peers at varying times, or may not be connected at all. All pipe resolution and communication is done within the scope of a peer group. That is, the output and input pipes must belong to the same peer group.

Pipes offer two modes of communication, point-to-point and propagate, as seen in the diagram below. JXSE 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. It is also possible for multiple peers to bind to a single input pipe.

- *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.

- *Secure Unicast Pipes*

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

Unidirectional pipes are a very-low level JXTA communication programming abstraction. It is recommended that most developers use the higher-level communication abstraction provided by the JxtaSocket and JxtaBiDipipe services described in the next section.

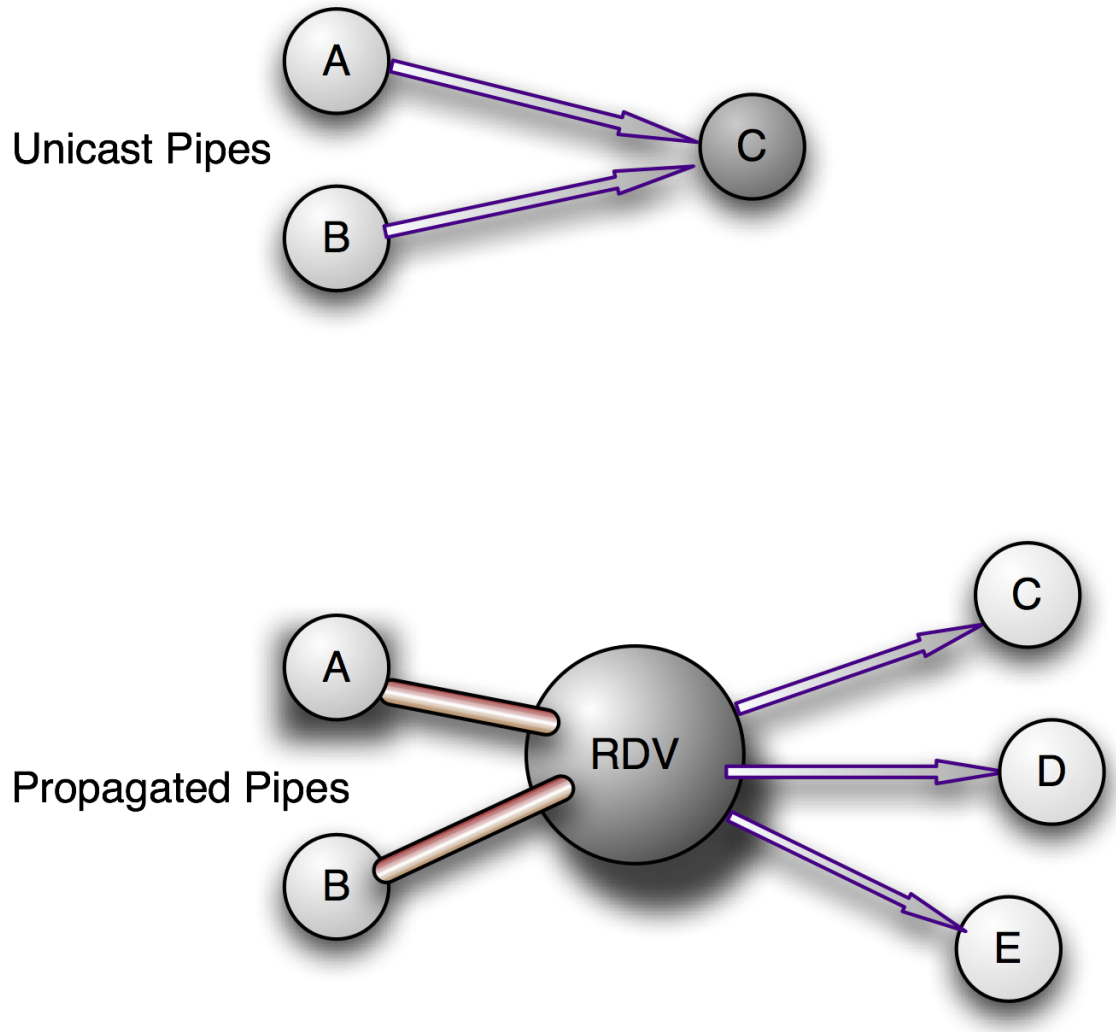


Figure 3: Unicast and Propagate Pipes

JxtaSocket and JxtaBiDiPipe

(Bidirectional reliable communication channels)

The basic JXTA pipes provide unidirectional, unreliable communication channels. In order to make pipes more useful to services and applications it is necessary to implement bidirectional and reliable communication channels on top of the pipe primitives. JXSE provides functionality to meet the level of service quality required by most applications:

- Reliability
- Ensures message sequencing
- Ensures delivery
- Exposes message and stream interfaces
- Security

- JxtaSocket and JxtaServerSocket :
  - Sub-class java.net.Socket and java.net.ServerSocket respectively
  - Are built on top of pipes, endpoint messengers, and the reliability library
  - Provide bidirectional, reliable and secure communication channels

    - Expose a stream based interface.
    - Provide configurable internal buffering and message chunking
    - Do not implement Nagle's algorithm, therefore streams must be flushed as needed
- JxtaBiDiPipe and JxtaServerPipe provides:
    - Are built on top of pipes, endpoint messengers, and the reliability library
    - Provide bidirectional, reliable, and secure communication channels
    - Expose a message based interface
    - Provide no message chunking (applications need to ensure message size does not exceed the platform message size limitation of 64K).

JxtaServerSocket and JxtaServerPipe expose an input pipe to process connection requests and negotiate communication parameters. JxtaSocket and JxtaBiDiPipe, on the other hand, bind to respective private dedicated pipes independent of the connection request pipe.

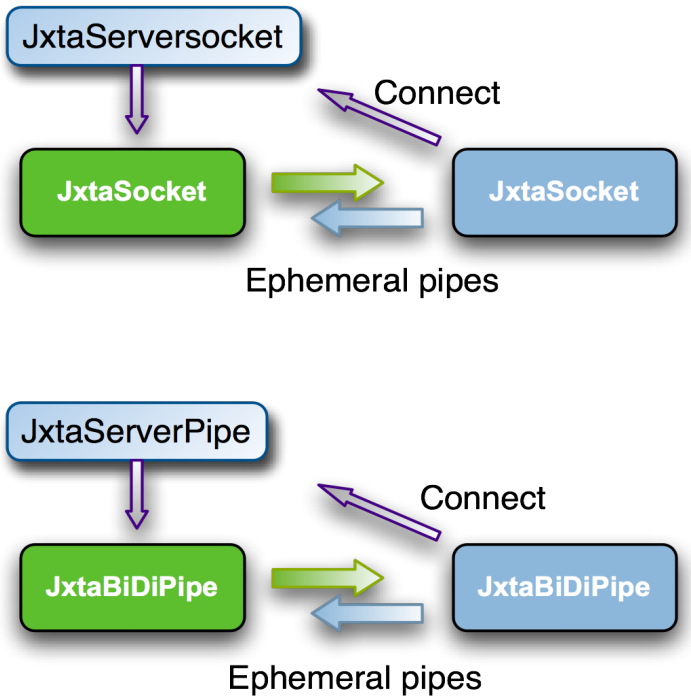


Figure 4: Pipe Connections

Advertisements

All JXTA network resources — such as peers, peer groups, pipes, and services — are represented as *advertisements*. Advertisements are language-neutral meta-data structures represented as XML documents. The JXTA protocols use advertisements to describe and publish the existence of a peer's 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's resources. 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).
- *ModuleSpecAdvertisement* — 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.
- *ModuleImplAdvertisement* – defines an implementation of a given module specification. It includes a 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 which are comprised of 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 5.

```
<?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
 </Name>
 </jxta:PipeAdvertisement>
```

Figure 5: A Pipe Advertisement

The complete specification of the JXTA advertisements is given in the *JXTA Protocols Specification* (see.http://specs.jxta.org/). Services or peer implementations may subtype any of the above advertisements to create their own application 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 meta-data such as credentials, certificates, digests, and public keys to JXTA messages, enabling these basic security requirements to be met. Message digests and signatures guarantee the data integrity of messages. Messages may also be encrypted and signed 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 it 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).

IDs

Peers, peer groups, pipes and other JXTA resources need to be uniquely identifiable. A JXTA ID uniquely identifies a resource and serves as a canonical way of referring to that resource. Currently, there are six types of JXTA entities which have JXTA ID types defined: peers, peer groups, pipes, content, 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
```

Every JXTA ID has an *ID Format.* The format describes how the ID was generated and how it may be manipulated by programs. Every ID indicates it's format immediately after the urn:jxta: prefix. There are two common JXTA ID Formats, `uuid` and `jxta`, though others exist. The `jxta` format is used for special common identifiers such as the IDs of the World Peer Group and the Network Peer Group. The `uuid` format is used for most other IDs. The `uuid` format provides randomly generated unique IDs and is based upon DCE GUID/UUIDs[2]. The portion of a JXTA ID which follows the ID Format is specific to each ID Format and is often opaque—you aren't meant to be able to decode it directly from the URI. In the prece

1 See IETF RFC 2141 for more information on URNs.

2 See "The Open Group - DCE 1.1 Remote Procedure Call (RPC) : Appendix A - Universal Unique Identifier" http://www.opengroup.org/onlinepubs/9629399/apdxa.htm

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, as a result, message routing between peers is non-deterministic. Peers may join or leave the network at any time; which results in ever changing routing information.

The only common aspect that various JXTA applications share is that they 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*

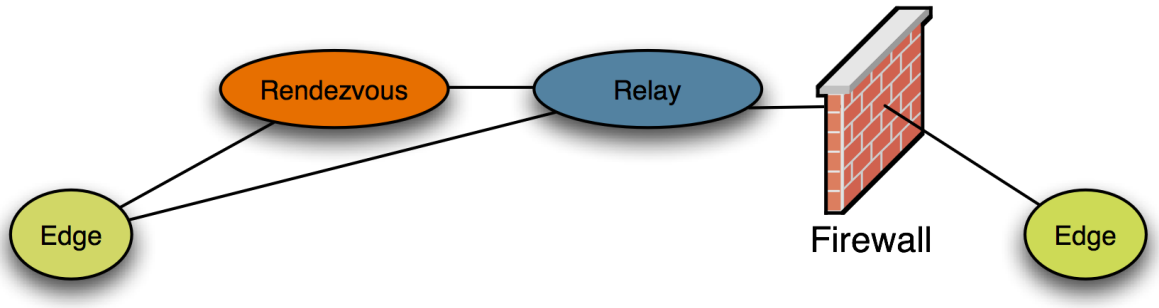
  A full-featured peer can send and receive messages and will typically cache advertisements. A simple peer replies to discovery requests with information found in its cached advertisements, but it does not forward any discovery requests. In any JXTA deployment most peers are likely to be edge peers.

- *Rendezvous peer*

  A rendezvous peer is an infrastructure peer, it aids other peers with message propagation, discovery of advertisements and routes, and most importantly it maintains a topology map of of other infrastructure peers, which then used for controlled propagation, and maintenance of the distributed hash table. Each peer group maintains its own set of rendezvous peers and may have as many rendezvous peers as needed. Edge peers send search and discovery requests to their rendezvous peer which in turn may forward requests it cannot answer to other known rendezvous peers using the topology mapped distributed hash table.

- *Relay peer*[3]

  A relay peer s an infrastructure peer, it aids non addressable (firewalled/NAT'd) peers with message relaying. A peer may request an in memory message box from a relay peer to facilitate message relaying whenever needed.



3 Relay peers were referred to as router peers in early JXTA documentation.

Shared Resource Distributed Index (SRDI)

The JXTA 2.0 J2SE platform supports a shared resource distributed index (SRDI) service to provide an 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 indexes 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 predefined 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 indexes 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). [4]

Queries

An example configuration is shown in Figure 6 below. 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.

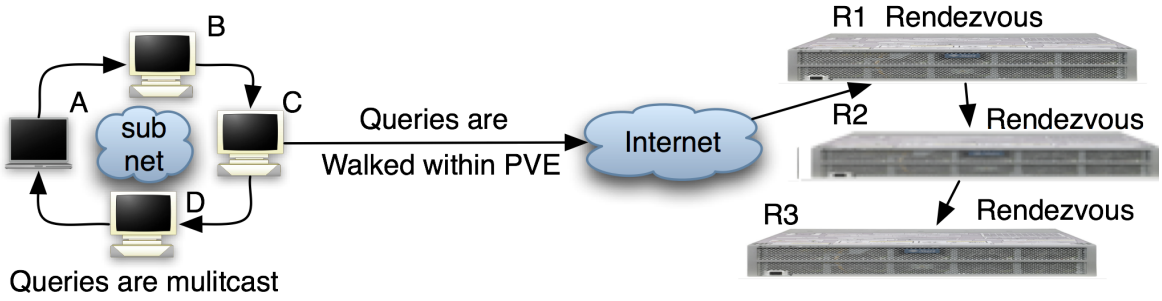


Figure 6: Request Propagation via Rendezvous Peers

Local network queries (i.e., within a subnet) are propagated to local network 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 network 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 cache, it will notify the peer that published the resource and that peer will respond directly to the requesting peer. The rendezvous is unable to respond directly to the querying peer because 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 nodes looking for a rendezvous that contains the index. A query path may be altered by a network map function to reduce the TTL of a query; A hop count is used to specify the maximum number of times the request is mapped/forwarded to avoid ping-pong effects which can occur in unstable or very dynamic networks. Once the query reaches the peer, it replies directly to the originator of the query.

4 See *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.

SRDI uses a SHA1 hash addressing scheme, where the 160 bit hash address space is divided amongst a ordered list of rendezvous nodes. When indexes are received they are hashed ,to determine their replication address, then replicated on their destination replica rendezvous.

Figure 7 below depicts a logical view of how the SRDI service works. Once Node A publishes a set of advertisements, a set of indexes in the form of an SRDI message is sent to its rendezvous, RDV1, where such Indexes are stored, then replicated (based on their hash mapping) on rendezvous 2, 3, and 4. Node C then issues a query for advertisement A, which is walked to rendezvous 2, then mapped to rendezvous 3, then finally forwarded node A.

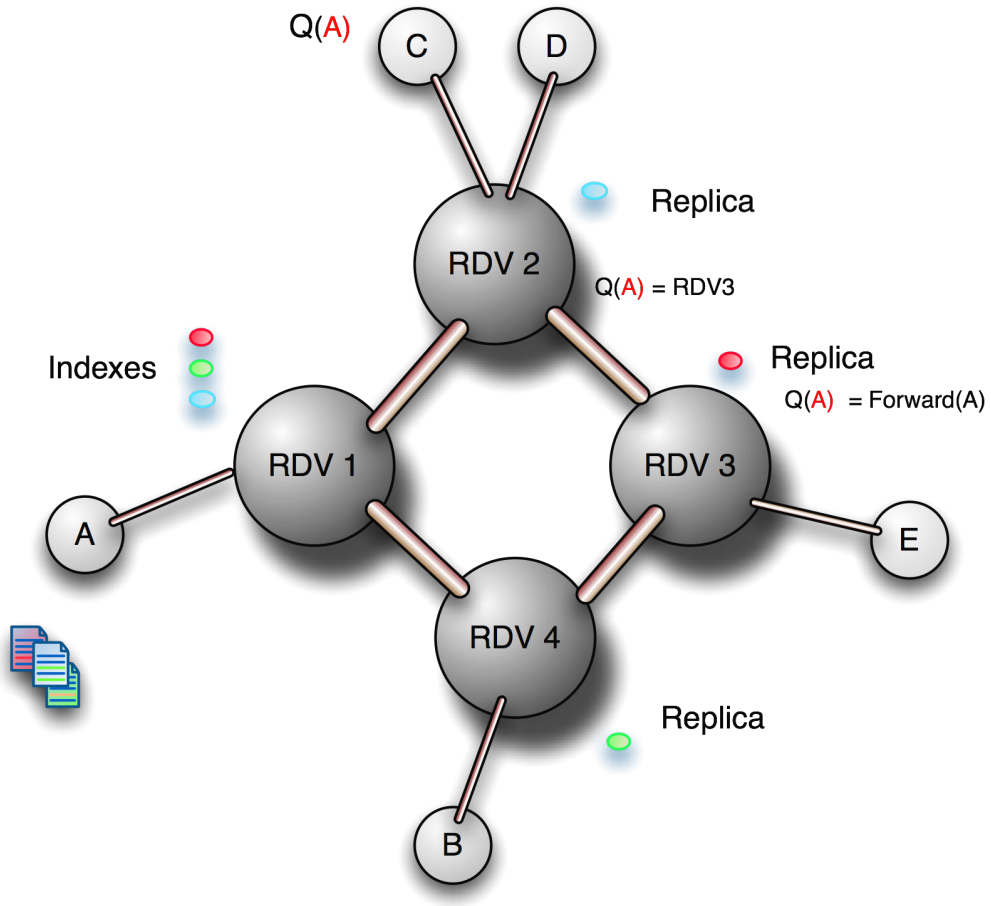


Figure 7: SRDI Operation

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 direct connection with a peer behind the firewall. The same is true for peers which are behind a NAT device.

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 a common transport (HTTP or TCP).
- The firewall, at the very least, has to allow outbound HTTP or TCP connections. 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.

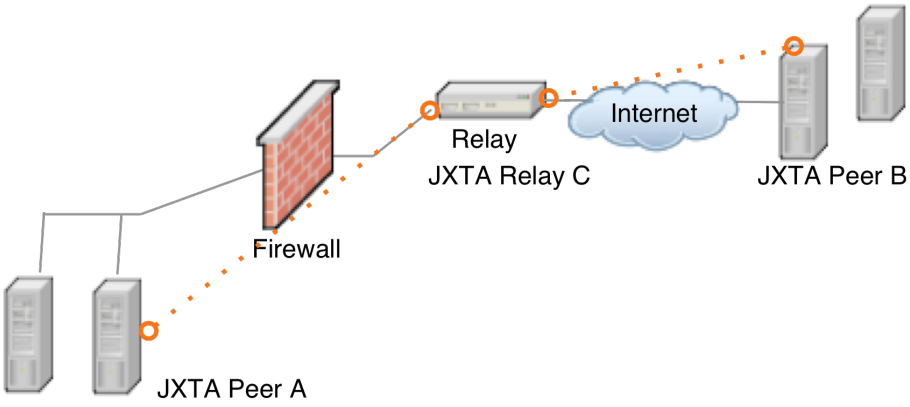


Figure 8: Message Routing Scenario Across a Firewall

Chapter 5:JXTA Protocols

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

There are six standard JXTA protocols[5]:

- *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 predefined 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)* — used by edge peers to resolve resources, propagate messages, and advertise local resources. used by rendezvous peers to organize with other rendezvous peers, share the distributed hash table address space, and propagate messages in controlled fashion (message walkers).

All of the standard 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. JXSE supports all six JXTA protocols. The Java SE API is used to access operations supported by these protocols, such as discovering peers or joining a peer group.

5 For a complete description of the JXTA protocols, please see the *JXTA Protocols Specification*, available for download from http://jxta-spec.dev.java.net.

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 the network. 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 define an alternate discovery service, the PDP is used to probe the network for advertisements.

There are multiple ways to discover distributed information. The current JXSE platform binding uses a combination of IP multicast to the local subnet and the use of a rendezvous network, which is a technique based on a rendezvous maintained DHT (Distributed Hash Table). Rendezvous nodes provide the mechanism of directing requests into the network to dynamically discover information. A node may be configured with a predefined set of rendezvous nodes. A node may also choose to bootstrap itself by dynamically locating rendezvous nodes or network resources in its local network via multicast messages.

Nodes generate discovery query messages to discover advertisements within a peer group. This message is enclosed within a resolver query contains the peer group credential of the probing node and identifies the probing peer to the message recipient. Messages can be sent to any node within a peer group.

A query is not guaranteed to result in any responses, or result in responses matching the requested threshold.

Peer Information Protocol

Once a node is located, it's 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 enforce the IT department's control of the 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 they 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 and 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 running 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's 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 enabling JXTA services to 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 indexes can be used to forward queries in the direction where the query is most likely to be answered and repropagates the messages to peers interested in those 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 (or 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, which supports 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) enables JXTA peers to send messages to remote peers without having a direct connection to the destination peer. The message will be passed through intermediary peers to reach it's final destination. ERP defines a query and response protocol which it uses to discover peer routing information and a message "envelope" that is attached to JXTA Messages describing the route a message should travel from one peer (the source) to another (the destination).
To send a message to another peer, the source peer first looks in its local cache to determine if it has a route to the destination peer. If it does not already have a route to the destination peer, it sends a route resolver query request asking for route information to the destination peer. Any peer receiving this route query will check to see if knows a route to the requested peer. If the peer does know of a valid route, it will respond to the route query with the route information for the desired destination peer. Any peer can query for route information and any peer within a peer group may offer route information and/or route messages destined for other peers. Relay peers typically cache route information.

Route query requests are sent by a peer to request route information for a destination peer. Route responses include the peer ID of the responder, the peer ID of the route's destination, and a semi-ordered sequence of peer IDs. The sequence of peer ID hops may provide a complete or partial route to the destination but at minimum contains one peer ID. In some cases the sequence may contain several alternative routes to the destination. A route can safely express alternatives because of the way in which routes are used.

The semi-ordered sequence of peer IDs provided in a Route Response provide information as to the path that a message may be forwarded in order to reach a destination peer. Each peer along the stated path may enhance and/or optimize the route the message takes based upon it's own knowledge. For example, if a peer receives a message containing a ten hop route to a destination peer but it is itself directly connected to the destination peer then it makes sense to forward the message directly rather than sending it along the long route. Similarly, a peer may shorten a route by using a shorter route it knows between any two hops in the route which is included with a message.

The Message routing procedure used by the Endpoint Routing Protocol is roughly as follows;

1. If I am originating the Message then check if I have a route to the destination. If I don't have a route to the destination, query for a route and wait for a route to be found. Eventually give up if no route is found.
2. If I am not originating the Message and do not have a route for the destination nor any peer listed in the route attached to the message then send a failure message to the peer from which I received the Message. JXTA peers do not currently generate route queries for messages they do not originate as this is too easily used to create distributed denial of service attacks (DDoS).
3. Remove my peer ID from the message route and all peer IDs which preceded mine in the Message route. Excluding all Peer Ids already in the Message Route, prepend the route I know to the destination peer to the Message route.
4. Starting at the last peer ID listed in the Message Route, check if I have a direct connection to any of the listed peer Ids. If so, remove all of the peer IDs before that peer in the Message route and forward the message to directly connected peer.
5. If no direct connection exists to any of the peer Ids listed in the Message route then forward the message to the first peer listed in the Message Route.

Rendezvous Protocol

The Rendezvous Protocol (RVP) is used for propagation of messages within a peer group. The RVP provides mechanisms which enable propagation of messages to be performed in a controlled and efficient way. The RVP is divided into three parts;

- The protocol used by the Rendezvous Peers to organize themselves, also known as the PeerView protocol.
- The protocol used by client peers to register interest in receiving propagation messages, a simple lease protocol.
- The protocol used for propagating messages to the peers which have expressed an interest in the destination address. The message propagation protocol is the only protocol which all participants must implement

Chapter 6:The Basics

This chapter discusses the JXTA development environment along with the steps required to compile and run the examples described in the guide. This chapter describes the following :

1. Getting started
   - System requirements
   - Accessing the on-line documentation
   - Downloading the Project JXTA binaries
   - Compiling JXTA technology code
   - Running JXTA technology applications
   - Configuring the JXTA environment
2. Creating various ID types
3. Creating a new Advertisement
4. Creating messages and message elements

Getting Started

System Requirements

The current Project JXTA JSE platform binding requires a platform that supports the Java Run-Time Environment (JRE) or Software Development Kit (SDK) 1.5.0 release or later. This environment is currently available for the Solaris Operating Environment, Microsoft Windows, Linux, Mac OS X and other operating systems.

The Java Standard Edition Platform JRE and SDK for Solaris SPARC/x86, Linux x86, and Microsoft Windows can be downloaded from:

http://java.sun.com/j2se/1.5.0/download.jsp

Accessing On-line Documentation

The On-line JavaDoc documentation for the JXTA JSE API is available at:
http://platform.jxta.org/nonav/java/api/overview-summary.html

Downloading Binaries

Download the Companion Tutorial 2.5 Programs[6] at http://www.jxta.org/ProgGuideExamples.zip. The compressed archive contains the JXTA platform and supporting libraries, sources and binaries of the tutorials covered in this guide, and run scripts.

You can download the latest JXTA builds at http://download.jxta.org/index.html. There are two types of builds available:

- *Release Builds* — These are thoroughly tested stable builds of the JXTA JSE 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/index.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.
- *Nightly Builds* — These are automated builds of the current JXTA "work in progress"; these builds are provided for developer testing and are not guaranteed to function correctly.

The source code for the various JXTA builds can also be downloaded allowing you to compile the Project JXTA source code yourself. Follow the directions on the Project JXTA Web page to download the source code and then build the binaries.

6 All of the covered tutorials are contained within the platform sources under http://www.jxta.org/Tutorials along with build tools.

Compiling JXTA Code

The application in this example, HelloWorld, requires the `jxta.jar` file for compilation. When you run the Java compiler (`javac`[7]), you need to include the `-classpath` option specifying the location of the `jxta.jar` file. You will need to provide the actual location of the `jxta.jar` file on your system.

Example compilation command (Windows systems):

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

Running JXTA Applications

When you enter the `java`[8] command to run the application, you need to include the `-classpath` option specifying the location of the required `jar` files. You will need to provide the location of the required `.jar` files.

Example tutorial run command (Windows systems):

```
C:> java -classpath .\lib\jxta.jar;.\lib\bcprov-jdk14.jar;.  IDTutorial
```

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 it.

The JXTA platform includes simple UI configurator. In most cases this default may be overridden by using a programmatic platform configurator such as the `NetworkConfigurator` class, or by using the `NetworkManager`, which abstracts configuration to one of the preset configurations:

1. Ad-Hoc: A node which is typically deployed in an ad-hoc network. The peer will not use any infrastructure peers (Rendezvous or Relay services).
2. Edge: In addition to supporting the Ad-Hoc behavior, an Edge node can attach to an infrastructure peer (a Rendezvous, Relay, or both).
3. Rendezvous: Provides network bootstrapping services, such as discovery, pipe resolution, etc.
4. Relay: Provides message relaying services, enabling cross firewall traversal.
5. Proxy: Provides JXME JXTA for J2ME proxy services.
6. Super: Provides the functionality of a Rendezvous, Relay, and Proxy node.

It is highly recommended that JXTA peers be configured through the NetworkManager, or NetworkConfigurator static configuration methods with the desired mode of operation such as `newEdgeConfiguration(storeHome)`. Once a configuration is created, it maybe tuned to the application requirements.

It's also worth noting the ability of the NetworkConfigurator to load a configuration given a URI. A feature that provides a systematic method for provisioning a node's PeerID and/or node certificates. For example, `load("http://web-provisioning.service.net/provision.cgi?name=test");`

7 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.

8 Again, see your Java documentation for specific details on running applications on your platform. Some environments use the -cp option to specify the classpath.

"Public JXTA Network" Configuration Setting

The JXTA development network which maybe used in some of the examples is provided to bootstrap demo applications and tutorials. It is not intended to be used as a production network. This configuration is to be used to test JXTA applications that run on peers located on multiple-subnets, firewalls and NATs.

JXTA and HTTP Proxies

In order enable JXTA peers to communicate across firewalls, JXTA supports the use of HTTP proxies. These are the same proxy servers used by web browsers. Some proxy server configurations may require authentication. In most cases JXTA will use the system HTTP proxy configuration settings or settings provided when Java was installed. You can also adjust the HTTP proxy settings using the Java Control Panel.

Do I need a Proxy?

If you need to use an HTTP proxy with your web browser then it is likely a proxy is needed for JXTA as well. Most web browsers have a preferences page containing the proxy information. (For "Internet Explorer" it is located in the "Connections/Lan Settings" dialog.). Some automatic browser proxy mechanisms can make it difficult to determine if you really are using a proxy. In these cases, ask a local expert or IT staff for help.

Configuring HTTP Proxy on the Java Command Line

If you start your application by running Java directly, you can specify the HTTP proxy parameters as part of the 'java' command line. The http proxy is specified by defining Java System Properties. For example:

```
% java -Dhttp.proxyHost=proxy.mycompany.com Dhttp.proxyPort=8080 ...
```

These settings would configure Java to use the proxy server proxy.mycompany.com at port 8080. You should use the values you copied from your web browser.

JXTA and Proxy Authentication

For HTTP proxies that require authentication, the following Java System Properties must be set:

```
% java -Djxta.proxy.user=usr -Djxta.proxy.password=pwd ..
```

Configuration information is stored in the file security information (user-name and password) is stored in the cm `./.cache/example_name/PlatformConfig;(./.cache/example_name/cm)`. The next time the application runs, this information is used to configure your peer.

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

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

Creating various ID types

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

URNs are used to express JXTA IDs. URNs[9] 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 strings.

An example JXTA peer ID is:

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

An example JXTA pipe ID is:

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

Every JXTA ID has an *ID Format.* The format describes how the ID was generated and how it may be manipulated by programs. Every ID indicates its format immediately after the urn:jxta: prefix. There are two common JXTA ID Formats, uuid and jxta, though others exist. The jxta format is used for special common identifiers such as the IDs of the World Peer Group and the Network Peer Group. The uuid format is is used for most other IDs. The uuid format provides randomly generated unique IDs and is based upon DCE GUID/UUIDs[10].

The following example illustrates how the various types of JXTA identifiers maybe constructed:

9 See IETF RFC 2141 for more information on URNs.

10 See "The Open Group - DCE 1.1 Remote Procedure Call (RPC) : Appendix A - Universal Unique Identifier" http://www.opengroup.org/onlinepubs/9629399/apdxa.htm

ID Tutorial source code

```
package tutorial.id;

import net.jxta.id.IDFactory;
import net.jxta.peer.PeerID;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.pipe.PipeID;

import java.io.UnsupportedEncodingException;
import java.security.MessageDigest;
import java.security.NoSuchAlgorithmException;
import java.text.MessageFormat;

/**
 * A simple and re-usable example of creating various JXTA IDs
 * <p/>
 * This is a two part tutorial :
 * <ol>
 * <li>Illustrates the creation of predictable ID's based upon the hash of a
 * provided string. This method provides an independent and deterministic
 * method for generating IDs. However using this method does require care
 * in the choice of the seed expression in order to prevent ID collisions
 * (duplicate IDs).</li>
 * <p/>
 * <li>New random ID's encoded with a specific GroupID.</li>
 * </ol>
 */
public class IDTutorial {
    private static final String SEED = "IDTuorial";

    /**
     * Returns a SHA1 hash of string.
     *
     * @param expression to hash
     * @return a SHA1 hash of string or {@code null} if the expression could
     *         not be hashed.
     */
    private static byte[] hash(final String expression) {
        byte[] result;
        MessageDigest digest;

        if (expression == null) {
            throw new IllegalArgumentException("Invalid null expression");
        }

        try {
            digest = MessageDigest.getInstance("SHA1");
        } catch (NoSuchAlgorithmException failed) {
            failed.printStackTrace(System.err);
            RuntimeException failure = new IllegalStateException("Could not get SHA-1
Message");
            failure.initCause(failed);
            throw failure;
        }

        try {
            byte[] expressionBytes = expression.getBytes("UTF-8");
            result = digest.digest(expressionBytes);
        } catch (UnsupportedEncodingException impossible) {
            RuntimeException failure =
                  new IllegalStateException("Could not encode expression as UTF8");
            failure.initCause(impossible);
            throw failure;
```

```
        }
        return result;
    }

    /**
     * Given a pipe name, it returns a PipeID who's value is chosen based upon that name.
     *
     * @param pipeName instance name
     * @param pgID     the group ID encoding
     * @return The pipeID value
     */
    public static PipeID createPipeID(PeerGroupID pgID, String pipeName) {
        String seed = pipeName + SEED;
        return IDFactory.newPipeID(pgID, hash(seed.toLowerCase()));
    }

    /**
     * Creates group encoded random PipeID.
     *
     * @param pgID the group ID encoding
     * @return The pipeID value
     */
    public static PipeID createNewPipeID(PeerGroupID pgID) {
        return IDFactory.newPipeID(pgID);
    }

    /**
     * Creates group encoded random PeerID.
     *
     * @param pgID the group ID encoding
     * @return The PeerID value
     */
    public static PeerID createNewPeerID(PeerGroupID pgID) {
        return IDFactory.newPeerID(pgID);
    }

    /**
     * Given a peer name generates a Peer ID who's value is chosen based upon that name.
     *
     * @param peerName instance name
     * @param pgID     the group ID encoding
     * @return The PeerID value
     */
    public static PeerID createPeerID(PeerGroupID pgID, String peerName) {
        String seed = peerName + SEED;
        return IDFactory.newPeerID(pgID, hash(seed.toLowerCase()));
    }

    /**
     * Creates group encoded random PeerGroupID.
     *
     * @param pgID the group ID encoding
     * @return The PeerGroupID value
     */
    public static PeerGroupID createNewPeerGroupID(PeerGroupID pgID) {
        return IDFactory.newPeerGroupID(pgID);
    }

    /**
     * Given a group name generates a Peer Group ID who's value is chosen based
     * upon that name.
     *
     * @param groupName group name encoding value
     * @return The PeerGroupID value
```

```
     */
    public static PeerGroupID createPeerGroupID(final String groupName) {
        //Use lower case to avoid any locale conversion inconsistencies
        return IDFactory.newPeerGroupID(PeerGroupID.defaultNetPeerGroupID,
                                        hash(SEED + groupName.toLowerCase()));
    }

    /**
     * Contructs and returns an string encoded Infrastructure PeerGroupID.
     *
     * @param groupName the string encoding
     * @return The infraPeerGroupID PeerGroupID
     */
    public static PeerGroupID createInfraPeerGroupID(String groupName) {
        return createPeerGroupID(groupName);
    }

    /**
     * Main method
     *
     * @param args command line arguments.  None defined
     */
    public static void main(String args[]) {
        PeerGroupID infra = createInfraPeerGroupID("infra");
        PeerID peerID = createPeerID(infra, "peer");
        PipeID pipeID = createPipeID(PeerGroupID.defaultNetPeerGroupID, "pipe");

        System.out.println(MessageFormat.format(
                        "\n\nAn infrastucture PeerGroupID: {0}", infra.toString()));
        System.out.println(MessageFormat.format(
               "PeerID with the above infra ID encoding: {0}", peerID.toString()));
        System.out.println(MessageFormat.format("PipeID with the default
defaultNetPeerGroupID encoding: {0}", pipeID.toString()));

        peerID = createNewPeerID(PeerGroupID.defaultNetPeerGroupID);
        pipeID = createNewPipeID(PeerGroupID.defaultNetPeerGroupID);
        PeerGroupID pgid = createNewPeerGroupID(PeerGroupID.defaultNetPeerGroupID);

        System.out.println(
          MessageFormat.format("\n\nNew PeerID created : {0}", peerID.toString()));
        System.out.println(
              MessageFormat.format("New PipeID created : {0}", pipeID.toString()));
        System.out.println(
           MessageFormat.format("New PeerGroupID created : {0}", pgid.toString()));
    }
}
```

Advertisements

Advertisements are core JXTA objects that are used to advertise Peers, PeerGroups, Services, Pipes or other JXTA resources. Advertisements provide a platform independent representation of core platform objects that can be exchanged between different platform implementations (Java, C, etc.).
Each Advertisement holds a document that represents the advertisement. Advertisements are typically represented as a text document (XML). The `Advertisement.getDocument(MimeMediaType)` method is used to generate representations of the advertisement. Different representations are available via mime type selection. Typical mime types are "text/xml" or "text/plain" that generate textual representations for the Advertisements.
Advertisement instances are created via `AdvertisementFactory` rather constructors on the Advertisement classes. All of public advertisements which are defined in the `net.jxta.protocol` package are abstract classes. For each abstract public advertisement there is also a private implementation class which handles the document creation and parsing tasks. These private implementations are registered with the AdvertisementFactory. New advertisement types must be registered with the AdverisementFactory as illustrated below, or by augmenting `platform/binding/java/impl/src/META-INF/services/net.jxta.document.Advertisement`.

The following example illustrates the creation of a new advertisement type, and the registration of the new advertisement with the advertisement factory through :

```
AdvertisementFactory.registerAdvertisementInstance(
      AdvertisementTutorial.getAdvertisementType(),new Advertisem entTu toria l.Ins tanti ator( ));
```

Advertisement Tutorial source

```
package tutorial.advertisement;

import net.jxta.document.*;
import net.jxta.id.ID;
import net.jxta.id.IDFactory;

import java.io.IOException;
import java.io.InputStream;
import java.io.Serializable;
import java.net.InetAddress;
import java.net.URI;
import java.net.URISyntaxException;
import java.net.UnknownHostException;
import java.util.Enumeration;
import java.util.logging.Logger;

/**
 * Simple Advertisement Tutorial creates a advertisment describing a system
 * <p/>
 * <pre>
 * &lt;?xml version="1.0"?>
 * &lt;!DOCTYPE jxta:System>
 * &lt;jxta:System xmlns:jxta="http://jxta.org">
 *   &lt;id>id&lt;/id>
 *   &lt;name>Device Name&lt;/name>
 *   &lt;ip>ip address&lt;/ip>
 *   &lt;hwarch>x86&lt;/hwarch>
 *   &lt;hwvendor>Sun MicroSystems&lt;/hwvendor>
 *   &lt;OSName>&lt;/OSName>
 *   &lt;OSVer>&lt;/OSVer>
 *   &lt;osarch>&lt;/osarch>
 *   &lt;sw>&lt;/sw>
 * &lt;/jxta:System>
 * </pre>
 */
public class AdvertisementTutorial extends Advertisement
        implements Comparable, Cloneable, Serializable {
    private String hwarch;
    private String hwvendor;
    private ID id = ID.nullID;
    private String ip;
    private String name;
    private String osname;
    private String osversion;
    private String osarch;
    private String inventory;

    private final static Logger LOG =
            Logger.getLogger(AdvertisementTutorial.class.getName());
    private final static String OSNameTag = "OSName";
    private final static String OSVersionTag = "OSVer";
    private final static String OSarchTag = "osarch";
    private final static String hwarchTag = "hwarch";
    private final static String hwvendorTag = "hwvendor";
    private final static String idTag = "ID";
    private final static String ipTag = "ip";
    private final static String nameTag = "name";
    private final static String swTag = "sw";

    /**
     * Indexable fields.  Advertisements must define the indexables, in order
     * to properly index and retrieve these advertisements locally and on the
```

```
     * network
     */
    private final static String[] fields = {idTag, nameTag, hwarchTag};

    /**
     * Default Constructor
     */
    public AdvertisementTutorial() {
    }

    /**
     * Construct from a StructuredDocument
     *
     * @param root Root element
     */
    public AdvertisementTutorial(Element root) {
        TextElement doc = (TextElement) root;

        if (!getAdvertisementType().equals(doc.getName())) {
            throw new IllegalArgumentException("Could not construct : " +
                    getClass().getName() + "from doc containing a " + doc.getName());
        }
        initialize(doc);
    }

    /**
     * Construct a doc from InputStream
     *
     * @param stream the underlying input stream.
     * @throws IOException if an I/O error occurs.
     */
    public AdvertisementTutorial(InputStream stream) throws IOException {
        StructuredTextDocument doc = (StructuredTextDocument)
                StructuredDocumentFactory
                        .newStructuredDocument(MimeMediaType.XMLUTF8, stream);
        initialize(doc);
    }

    /**
     * Sets the hWArch attribute of the AdvertisementTutorial object
     *
     * @param hwarch The new hWArch value
     */
    public void setHWArch(String hwarch) {
        this.hwarch = hwarch;
    }

    /**
     * Sets the OSArch attribute of the AdvertisementTutorial object
     *
     * @param osarch The new hWArch value
     */
    public void setOSArch(String osarch) {
        this.osarch = osarch;
    }

    /**
     * Sets the hWVendor attribute of the AdvertisementTutorial object
     *
     * @param hwvendor The new hWVendor value
     */
    public void setHWVendor(String hwvendor) {
        this.hwvendor = hwvendor;
    }
```

```
    /**
     * sets the unique id
     *
     * @param id The id
     */
    public void setID(ID id) {
        this.id = (id == null ? null : id);
    }

    /**
     * Sets the iP attribute of the AdvertisementTutorial object
     *
     * @param ip The new iP value
     */
    public void setIP(String ip) {
        this.ip = ip;
    }

    /**
     * Sets the name attribute of the AdvertisementTutorial object
     *
     * @param name The new name value
     */
    public void setName(String name) {
        this.name = name;
    }

    /**
     * Sets the oSName attribute of the AdvertisementTutorial object
     *
     * @param osname The new oSName value
     */
    public void setOSName(String osname) {
        this.osname = osname;
    }

    /**
     * Sets the oSVersion attribute of the AdvertisementTutorial object
     *
     * @param osversion The new oSVersion value
     */
    public void setOSVersion(String osversion) {
        this.osversion = osversion;
    }

    /**
     * Sets the SWInventory attribute of the AdvertisementTutorial object
     *
     * @param inventory the software inventory of the system
     */
    public void setSWInventory(String inventory) {
        this.inventory = inventory;
    }

    /**
     * {@inheritDoc}
     *
     * @param asMimeType Document encoding
     * @return The document value
     */
    @Override
    public Document getDocument(MimeMediaType asMimeType) {
        StructuredDocument adv =
```

```
StructuredDocumentFactory.newStructuredDocument(asMimeType,
                getAdvertisementType());
        if (adv instanceof Attributable) {
            ((Attributable) adv).addAttribute("xmlns:jxta", "http://jxta.org");
        }
        Element e;
        e = adv.createElement(idTag, getID().toString());
        adv.appendChild(e);
        e = adv.createElement(nameTag, getName().trim());
        adv.appendChild(e);
        e = adv.createElement(OSNameTag, getOSName().trim());
        adv.appendChild(e);
        e = adv.createElement(OSVersionTag, getOSVersion().trim());
        adv.appendChild(e);
        e = adv.createElement(OSarchTag, getOSArch().trim());
        adv.appendChild(e);
        e = adv.createElement(ipTag, getIP().trim());
        adv.appendChild(e);
        e = adv.createElement(hwarchTag, getHWArch().trim());
        adv.appendChild(e);
        e = adv.createElement(hwvendorTag, getHWVendor().trim());
        adv.appendChild(e);
        e = adv.createElement(swTag, getSWInventory().trim());
        adv.appendChild(e);
        return adv;
    }

    /**
     * Gets the hWArch attribute of the AdvertisementTutorial object
     *
     * @return The hWArch value
     */
    public String getHWArch() {
        return hwarch;
    }

    /**
     * Gets the OSArch attribute of the AdvertisementTutorial object
     *
     * @return The OSArch value
     */
    public String getOSArch() {
        return osarch;
    }

    /**
     * Gets the hWVendor attribute of the AdvertisementTutorial object
     *
     * @return The hWVendor value
     */
    public String getHWVendor() {
        return hwvendor;
    }

    /**
     * returns the id of the device
     *
     * @return ID the device id
     */
    @Override
    public ID getID() {
        return (id == null ? null : id);
    }
```

```
    /**
     * Gets the IP attribute of the AdvertisementTutorial object
     *
     * @return The IP value
     */
    public String getIP() {
        return ip;
    }

    /**
     * Gets the name attribute of the AdvertisementTutorial object
     *
     * @return The name value
     */
    public String getName() {
        return name;
    }

    /**
     * Gets the OSName attribute of the AdvertisementTutorial object
     *
     * @return The OSName value
     */
    public String getOSName() {
        return osname;
    }

    /**
     * Gets the Software Inventory text element
     *
     * @return The Inventory value
     */
    public String getSWInventory() {
        if (inventory == null) {
            inventory = "";
        }
        return inventory;
    }

    /**
     * Gets the OSVersion attribute of the AdvertisementTutorial object
     *
     * @return The OSVersion value
     */
    public String getOSVersion() {
        return osversion;
    }

    /**
     * Process an individual element from the document.
     *
     * @param elem the element to be processed.
     * @return true if the element was recognized, otherwise false.
     */
    protected boolean handleElement(TextElement elem) {
        if (elem.getName().equals(idTag)) {
            try {
                URI id = new URI(elem.getTextValue());
                setID(IDFactory.fromURI(id));
            } catch (URISyntaxException badID) {
                throw new IllegalArgumentException("unknown ID format in
advertisement: " +
                        elem.getTextValue());
            }
```

```
            catch (ClassCastException badID) {
                throw new IllegalArgumentException("Id is not a known id type: " +
                        elem.getTextValue());
            }
            return true;
        }
        if (elem.getName().equals(nameTag)) {
            setName(elem.getTextValue());
            return true;
        }
        if (elem.getName().equals(OSNameTag)) {
            setOSName(elem.getTextValue());
            return true;
        }
        if (elem.getName().equals(OSVersionTag)) {
            setOSVersion(elem.getTextValue());
            return true;
        }
        if (elem.getName().equals(OSarchTag)) {
            setOSArch(elem.getTextValue());
            return true;
        }
        if (elem.getName().equals(ipTag)) {
            setIP(elem.getTextValue());
            return true;
        }
        if (elem.getName().equals(hwarchTag)) {
            setHWArch(elem.getTextValue());
            return true;
        }
        if (elem.getName().equals(hwvendorTag)) {
            setHWVendor(elem.getTextValue());
            return true;
        }
        if (elem.getName().equals(swTag)) {
            setSWInventory(elem.getTextValue());
            return true;
        }
        // element was not handled
        return false;
    }

    /**
     * Intialize a System advertisement from a portion of a structured document.
     *
     * @param root document root
     */
    protected void initialize(Element root) {
        if (!TextElement.class.isInstance(root)) {
            throw new IllegalArgumentException(getClass().getName() +
                    " only supports TextElement");
        }
        TextElement doc = (TextElement) root;
        if (!doc.getName().equals(getAdvertisementType())) {
            throw new IllegalArgumentException("Could not construct : "
                    + getClass().getName() + "from doc containing a " +
                    doc.getName());
        }
        Enumeration elements = doc.getChildren();
        while (elements.hasMoreElements()) {
            TextElement elem = (TextElement) elements.nextElement();
            if (!handleElement(elem)) {
                LOG.warning("Unhandleded element \'" + elem.getName() + "\' in " +
                        doc.getName());
```

```
            }
        }
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public final String[] getIndexFields() {
        return fields;
    }

    /**
     * {@inheritDoc}
     */
    @Override
    public boolean equals(Object obj) {
        if (this == obj) {
            return true;
        }
        if (obj instanceof AdvertisementTutorial) {
            AdvertisementTutorial adv = (AdvertisementTutorial) obj;
            return getID().equals(adv.getID());
        }
        return false;
    }

    /**
     * {@inheritDoc}
     */
    public int compareTo(Object other) {
        return getID().toString().compareTo(other.toString());
    }

    /**
     * All messages have a type (in xml this is &#0033;doctype) which
     * identifies the message
     *
     * @return String "jxta:AdvertisementTutorial"
     */
    public static String getAdvertisementType() {
        return "jxta:AdvertisementTutorial";
    }

    /**
     * Instantiator
     */
    public static class Instantiator implements AdvertisementFactory.Instantiator {

        /**
         * Returns the identifying type of this Advertisement.
         *
         * @return String the type of advertisement
         */
        public String getAdvertisementType() {
            return AdvertisementTutorial.getAdvertisementType();
        }

        /**
         * Constructs an instance of <CODE>Advertisement</CODE> matching the
         * type specified by the <CODE>advertisementType</CODE> parameter.
         *
         * @return The instance of <CODE>Advertisement</CODE> or null if it
         *         could not be created.
```

```
         */
        public Advertisement newInstance() {
            return new AdvertisementTutorial();
        }

        /**
         * Constructs an instance of <CODE>Advertisement</CODE> matching the
         * type specified by the <CODE>advertisementType</CODE> parameter.
         *
         * @param root Specifies a portion of a StructuredDocument which will
         *             be converted into an Advertisement.
         * @return The instance of <CODE>Advertisement</CODE> or null if it
         *         could not be created.
         */
        public Advertisement newInstance(net.jxta.document.Element root) {
            return new AdvertisementTutorial(root);
        }
    }

    /**
     * Main method
     *
     * @param args command line arguments.  None defined
     */
    public static void main(String args[]) {

        // The following step is required and only need to be done once,
        // without this step the AdvertisementFactory has no means of
        // associating an advertisement name space with the proper obect
        // in this cast the AdvertisementTutorial
        AdvertisementFactory.registerAdvertisementInstance(
                AdvertisementTutorial.getAdvertisementType(),
                new AdvertisementTutorial.Instantiator());

        AdvertisementTutorial advTutorial = new AdvertisementTutorial();
        advTutorial.setID(ID.nullID);
        advTutorial.setName("AdvertisementTutorial");
        try {
            advTutorial.setIP(InetAddress.getLocalHost().getHostAddress());
        } catch (UnknownHostException ignored) {
            //ignored
        }
        advTutorial.setOSName(System.getProperty("os.name"));
        advTutorial.setOSVersion(System.getProperty("os.version"));
        advTutorial.setOSArch(System.getProperty("os.arch"));
        advTutorial.setHWArch(System.getProperty("HOSTTYPE",
                System.getProperty("os.arch")));
        advTutorial.setHWVendor(System.getProperty("java.vm.vendor"));
        System.out.println(advTutorial.toString());
    }
}
```

Message and Message Element tutorial source

```
package tutorial.message;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.XMLDocument;
import net.jxta.endpoint.ByteArrayMessageElement;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.Message.ElementIterator;
import net.jxta.endpoint.MessageElement;
import net.jxta.endpoint.StringMessageElement;
import net.jxta.endpoint.TextDocumentMessageElement;
import net.jxta.endpoint.WireFormatMessage;
import net.jxta.endpoint.WireFormatMessageFactory;
import net.jxta.id.IDFactory;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.pipe.PipeService;
import net.jxta.protocol.PipeAdvertisement;

import java.io.ByteArrayOutputStream;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.ObjectInputStream;
import java.io.ObjectOutputStream;
import java.io.RandomAccessFile;
import java.util.zip.GZIPInputStream;
import java.util.zip.GZIPOutputStream;

/**
 * A simple and re-usable example of manipulating JXATA Messages. Included in
 * this tutorial are:
 * <p/>
 * <ul>
 * <li>Adding and reading {@code String}, {@code int} and {@code long} with Message elements
 * <li>Adding and reading Java {@code Object} with Message Elements.
 * <li>Adding and reading byte arrays with Message Elements.</li>
 * <li>Adding and reading JXTA Advertisements with Message Elements.</li>
 * <li>Compressing message element content with gzip.</li>
 * </ul>
 */
public class MessageTutorial {
    private final static MimeMediaType GZIP_MEDIA_TYPE =
                            new MimeMediaType("application/gzip").intern();

    /**
     * Adds a String to a Message as a StringMessageElement
     *
     * @param message   The message to add to
     * @param nameSpace The namespace of the element to add. a null value assumes
default namespace.
     * @param elemName  Name of the Element.
     * @param string    The string to add
     */
    public static void addStringToMessage(Message message,
                              String nameSpace, String elemName, String string) {
        message.addMessageElement(nameSpace,
                new StringMessageElement(elemName,
                        string,
                        null));
    }
```

```
    /**
     * Adds a long to a message
     *
     * @param message   The message to add to
     * @param nameSpace The namespace of the element to add. a null value assumes
default namespace.
     * @param elemName  Name of the Element.
     * @param data      The feature to be added to the LongToMessage attribute
     */
    public static void addLongToMessage(Message message,
                          String nameSpace, String elemName, long data) {
        message.addMessageElement(nameSpace,
                new StringMessageElement(elemName,
                        Long.toString(data),
                        null));
    }

    /**
     * Adds a int to a message
     *
     * @param message   The message to add to
     * @param nameSpace The namespace of the element to add. a null value assumes
default namespace.
     * @param elemName  Name of the Element.
     * @param data      The feature to be added to the IntegerToMessage attribute
     */
    public static void addIntegerToMessage(Message message, String nameSpace, String
elemName, int data) {
        message.addMessageElement(nameSpace,
                new StringMessageElement(elemName,
                        Integer.toString(data),
                        null));
    }

    /**
     * Adds an byte array to a message
     *
     * @param message   The message to add to
     * @param nameSpace The namespace of the element to add. a null value assumes
default namespace.
     * @param elemName  Name of the Element.
     * @param data      the byte array
     * @param compress  indicates whether to use GZIP compression
     * @throws IOException if an io error occurs
     */
    public static void addByteArrayToMessage(Message message, String nameSpace,
String elemName, byte[] data, boolean compress) throws IOException {
        byte[] buffer = data;
        MimeMediaType mimeType = MimeMediaType.AOS;
        if (compress) {
            ByteArrayOutputStream outStream = new ByteArrayOutputStream();
            GZIPOutputStream gos = new GZIPOutputStream(outStream);
            gos.write(data, 0, data.length);
            gos.finish();
            gos.close();
            buffer = outStream.toByteArray();
            mimeType = GZIP_MEDIA_TYPE;
        }
        message.addMessageElement(nameSpace,
                new ByteArrayMessageElement(elemName,
                        mimeType,
                        buffer,
                        null));
    }
```

```
    /**
     * Adds an Object to message within the specified name space and with the
specified element name
     * @param message   the message to add the object to
     * @param nameSpace the name space to add the object under
     * @param elemName  the given element name
     * @param object    the object
     * @throws IOException if an io error occurs
     */
    public static void addObjectToMessage(Message message, String nameSpace,
                                String elemName, Object object) throws IOException {
        ByteArrayOutputStream bos = new ByteArrayOutputStream();
        ObjectOutputStream oos = new ObjectOutputStream(bos);

        oos.writeObject(object);
        oos.close();
        bos.close();
        addByteArrayToMessage(message, nameSpace, elemName, bos.toByteArray(), false);
    }

    /**
     * Returns a String from a message
     *
     * @param message   The message to retrieve from
     * @param nameSpace The namespace of the element to get.
     * @param elemName  Name of the Element.
     * @return The string value or {@code null} if there was no element matching the
specified name.
     */
    public static String getStringFromMessage(Message message,
                                              String nameSpace, String elemName) {
        MessageElement me = message.getMessageElement(nameSpace, elemName);

        if (null != me) {
            return me.toString();
        } else {
            return null;
        }
    }

    /**
     * Returns an long from a message
     *
     * @param message   The message to retrieve from
     * @param nameSpace The namespace of the element to get.
     * @param elemName  Name of the Element.
     * @return The long value
     * @throws NumberFormatException If the String does not contain a parsable int.
     */
    public static long getLongFromMessage(Message message,
                  String nameSpace, String elemName) throws NumberFormatException {
        String longStr = getStringFromMessage(message, nameSpace, elemName);
        if (null != longStr) {
            return Long.parseLong(longStr);
        } else {
            throw new NumberFormatException("No such Message Element.");
        }
    }

    /**
     * Returns an int from a message
     *
     * @param message   The message to retrieve from
```

```
     * @param nameSpace The namespace of the element to get.
     * @param elemName  Name of the Element.
     * @return The int value
     * @throws NumberFormatException If the String does not contain a parsable long.
     */
    public static int getIntegerFromMessage(Message message, String nameSpace,
                                    String elemName) throws NumberFormatException {
        String intStr = getStringFromMessage(message, nameSpace, elemName);

        if (null != intStr) {
            return Integer.parseInt(intStr);
        } else {
            throw new NumberFormatException("No such Message Element.");
        }
    }

    /**
     * Returns an InputStream for a byte array
     *
     * @param message   The message to retrieve from
     * @param nameSpace The namespace of the element to get.
     * @param elemName  Name of the Element.
     * @return The {@code InputStream} or {@code null} if the message has no such
element, String elemName) throws IOException {
     * @throws IOException if an io error occurs
     */
    public static InputStream getInputStreamFromMessage(Message message,
                            String nameSpace, String elemName) throws IOException {
        InputStream result = null;
        MessageElement element = message.getMessageElement(nameSpace, elemName);

        if (null == element) {
            return null;
        }

        if (element.getMimeType().equals(GZIP_MEDIA_TYPE)) {
            result = new GZIPInputStream(element.getStream());
        } else if (element.getMimeType().equals(MimeMediaType.AOS)) {
            result = element.getStream();
        }
        return result;
    }

    /**
     * Reads a single Java Object from a Message.
     *
     * @param message   The message containing the object.
     * @param nameSpace The name space of the element containing the object.
     * @param elemName  The name of the element containing the object.
     * @return The Object or {@code null} if the Message contained no such element.
     * @throws IOException if an io error occurs
     * @throws ClassNotFoundException if an object could not constructed from the message element
     */
    public static Object getObjectFromMessage(Message message, String nameSpace,
String elemName) throws IOException, ClassNotFoundException {
        InputStream is = getInputStreamFromMessage(message, nameSpace, elemName);

        if (null == is) {
            return null;
        }
        ObjectInputStream ois = new ObjectInputStream(is);
        return ois.readObject();
    }
```

```
    /**
     * Prints message element names and content and some stats
     *
     * @param msg     message to print
     * @param verbose indicates whether to print elment content
     */
    public static void printMessageStats(Message msg, boolean verbose) {
        try {
            System.out.println("------------------Begin Message----------------------");
            WireFormatMessage serialed = WireFormatMessageFactory.toWire(
                    msg,
                    new MimeMediaType("application/x-jxta-msg"), null);
            System.out.println("Message Size :" + serialed.getByteLength());

            ElementIterator it = msg.getMessageElements();
            while (it.hasNext()) {
                MessageElement el = it.next();
                System.out.println("Element : " + it.getNamespace() + " :: " +
el.getElementName());
                if (verbose) {
                    System.out.println("[" + el + "]");
                }
            }
            System.out.println("------------------End Message----------------------");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * Illustrates adding and retrieving a String to and from a Message
     */
    public static void stringExample() {
        Message message = new Message();
        addStringToMessage(message, "TutorialNameSpace", "String Test", "This is a test");
        printMessageStats(message, true);
        System.out.println("String Value :" +
                getStringFromMessage(message, "TutorialNameSpace", "String Test"));
    }

    /**
     * Illustrates adding and retrieving a long to and from a Message
     */
    public static void longExample() {
        Message message = new Message();
        addLongToMessage(message, "TutorialNameSpace", "long test", Long.MAX_VALUE);
        printMessageStats(message, true);
        System.out.println("long Value :" +
              getLongFromMessage(message, "TutorialNameSpace", "long test"));
    }

    /**
     * Illustrates adding and retrieving an integer to and from a Message
     */
    public static void intExample() {
        Message message = new Message();
        addIntegerToMessage(message, "TutorialNameSpace", "int test", Integer.MAX_VALUE);
        printMessageStats(message, true);
        System.out.println("int Value :" +
             getIntegerFromMessage(message, "TutorialNameSpace", "int test"));
    }

    /**
     * Illustrates adding and retrieving byte-array to and from a Message
```

```
     */
    public static void byteArrayExample() {
        Message message = new Message();
        try {
            File file = new File("message.tst");
            file.createNewFile();
            RandomAccessFile raf = new RandomAccessFile(file, "rw");
            raf.setLength(1024 * 4);
            int size = 4096;
            byte[] buf = new byte[size];
            raf.read(buf, 0, size);
            addByteArrayToMessage(message, "TutorialNameSpace", "byte test", buf, true);
            printMessageStats(message, true);
            InputStream is = getInputStreamFromMessage(message,
                                                "TutorialNameSpace", "byte test");
            int count = 0;
            while (is.read() != -1) {
                count++;
            }
            System.out.println("Read " + count + " byte back");
        } catch (IOException io) {
            io.printStackTrace();
        }
    }

    /**
     * Illustrates adding and retrieving advertisements to and from a Message
     */
    public static void xmlDocumentExample() {
        Message message = new Message();
        PipeAdvertisement pipeAdv = (PipeAdvertisement)
                AdvertisementFactory.newAdvertisement(
                        PipeAdvertisement.getAdvertisementType());
        pipeAdv.setPipeID(IDFactory.newPipeID(PeerGroupID.defaultNetPeerGroupID));
        pipeAdv.setType(PipeService.UnicastType);
        message.addMessageElement("MESSAGETUT", new
TextDocumentMessageElement("MESSAGETUT",
                (XMLDocument) pipeAdv.getDocument(MimeMediaType.XMLUTF8), null));
        MessageElement msgElement = message.getMessageElement("MESSAGETUT", "MESSAGETUT");
        try {
            XMLDocument asDoc = (XMLDocument)
StructuredDocumentFactory.newStructuredDocument(msgElement.getMimeType(),
                                                        msgElement.getStream());
            PipeAdvertisement newPipeAdv = (PipeAdvertisement)
                                    AdvertisementFactory.newAdvertisement(asDoc);
            System.out.println(newPipeAdv.toString());
        } catch (IOException e) {
            // This is thrown if the message element could not be read.
            e.printStackTrace();
        } catch (IllegalArgumentException e) {
            // This is thrown if the document or advertisement is invalid (illegal
            // values, missing tags, etc.)
            e.printStackTrace();
        }
    }


    /**
     * Main method
     *
     * @param args command line arguments.  None defined
     */
    public static void main(String args[]) {
        stringExample();
```

```
        longExample();
        intExample();
        byteArrayExample();
        xmlDocumentExample();
    }
}
```

Chapter 7:Programming with JXTA

HelloWorld Example

This example illustrates starting and stopping the JXTA platform. The application controls configuration, startup and stopping of the JXTA platform through the NetworkManager.

Example output: HelloWorld.

```
Starting JXTA
JXTA Started
Waiting for a rendezvous connection
Connected : true
Stopping JXTA
```

Hello World Example:

The code for this example begins on page 54. We define a single class, HelloWorld, with one class variable:

- `NetworkManager manager` — the NetworkManager, for configuring, starting and stopping the JXTA platform, and one method:
- static public void main()— main routine; configures and starts the platform, waits 2 minutes for a rendezvous connection then stops the platform.

main()

```
manager = new NetworkMan ager( Netwo rkMan ager. MODE. EDGE, "HelloWo rld" );
```

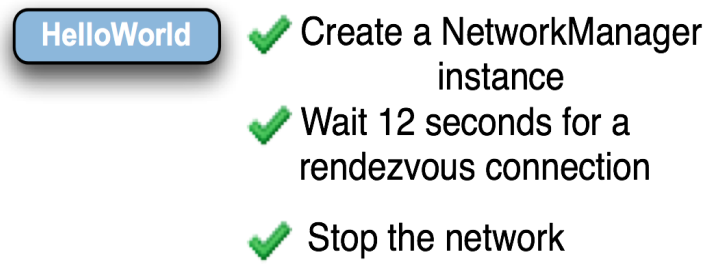
This configures the peer as edge node, with a peer name of *"HelloWo rld"* , this call only creates the default configuration for the JXTA platform. The application can customize the configuration and make additional configuration tunings if so desired. This can be done by obtaining the NetworkConfigurator object from the NetworkManager :

```
NetworkC onfig urato r configur ator = manager. getCo nfigu rator ();
```

Running the HelloWorld Example

When the application completes, you can inspect the various files and subdirectories that were created in the `./.cache /HelloWorld` 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).



Source Code: HelloWorld

```
package tutorial.helloworld;

import net.jxta.platform.NetworkManager;
import java.text.MessageFormat;

/**
 * A example of starting and stopping JXTA
 */
public class HelloWorld {

    /**
     * Main method
     *
     * @param args none defined
     */
    public static void main(String args[]) {
        NetworkManager manager = null;
        try {
            manager = new NetworkManager(NetworkManager.ConfigMode.EDGE, "HelloWorld");
            System.out.println("Starting JXTA");
            manager.startNetwork();
            System.out.println("JXTA Started");
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }
        System.out.println("Waiting for a rendezvous connection");
        boolean connected = manager.waitForRendezvousConnection(12000);
        System.out.println(MessageFormat.format("Connected :{0}", connected));
        System.out.println("Stopping JXTA");
        manager.stopNetwork();
    }
}
```

Chapter 8: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.

shows example output when this application is run:

Example output: Peer discovery example.

```
Sending a Discovery Message
Sending a Discovery Message
Got a Discovery Response [3 elements] from peer : unknown
 Peer name = node1
 Peer name = node2
 Peer name = JXTA.ORG 237
 Sending a Discovery message
Got a Discovery Response [3 elements] from peer : unknown
 Peer name = node3
 Peer name = node4
 Peer name = node5
```

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 are most likely the only peer on the network, or the peer is not configured correctly within its environment. A Rendezvous node is needed when nodes are spread across sub-nets, where multicast is not available. A relay node is needed when nodes are separated by firewalls or NATs.

Discovery Service

The JXTA Discovery Service provides an asynchronous interface for discovering peer, peer group, pipe, and service advertisements. Advertisements are stored in a persistent local cache (the $JXTA_HOME which defaults to `./.jxta/cm` directory). When a peer restarts, the same cache is referenced. Within the `./.jxta/cm` directory, subdirectories are created for each peer group that is joined.

- `./.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

Advertisements can retrieved from the local cache by invoking the getLocalAdvertisements() method, and can discover additional advertisements by invoking getRemoteAdvertisements(), and passing an optional discovery callback listener. A call to getRemoteAdvertisements() method maybe directed at specific node by specifying a node's PeerID, if none are specified, then the Discovery service will make a best attempt at directing the query based on it's configuration and environment. An undirected query will normally be propagated over all defined transports, and to the rendezvous connection if any.

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 Discovery Service can be used to publish advertisements. This is discussed in more detail in the "Creating Peer Groups and Publishing Advertisement" tutorial.

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"

DiscoveryClient

This is a two part example, a client and a server, and uses the DisoveryListener interface to receive asynchronous notification of discovery events. [The code for this example begins on page 60]. 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) and four methods:

- `public void start()` — starts the tutorial
- `public void stop()` — stops the tutorial
- `public void discoveryEvent(DiscoveryEvent ev)` — Discovery callback method, which invoked for every `DiscoveryResponse` message received
- `static public void main()` — main routine

main() method

Initializes the class, which configures, and starts the JXTA platform. Then invokes the start() method to start the tutorial.

start() method

The start() method first adds the calling object as a DiscoveryListener for DiscoveryResponse events :

```
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 node receives a Discovery Response message.

Next, the start() 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 discovery requests 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 a wildcard (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, 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 :

```
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 :

```
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 :

```
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 :

```
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) :

```
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 :

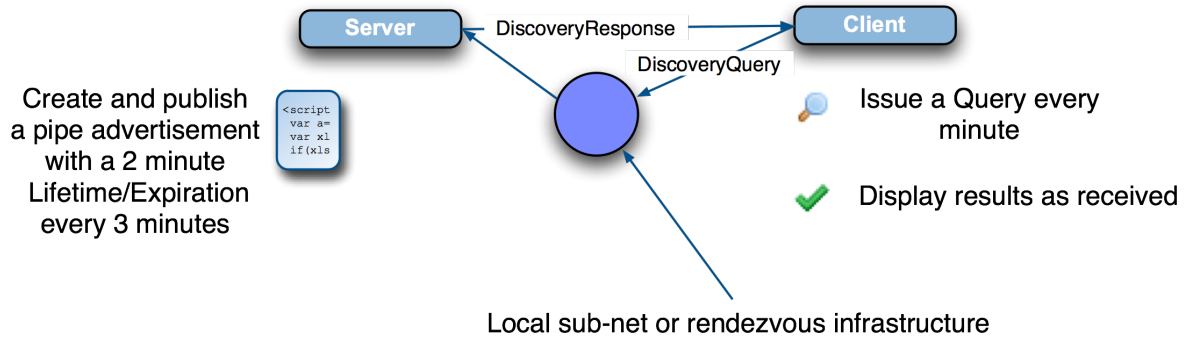
```
Enumeration en = res.getAdvertisements();
```

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

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

main()

The main() method 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: DiscoveryClient

```
package tutorial.discovery;

import net.jxta.discovery.DiscoveryEvent;
import net.jxta.discovery.DiscoveryListener;
import net.jxta.discovery.DiscoveryService;
import net.jxta.document.Advertisement;
import net.jxta.peergroup.PeerGroup;
import net.jxta.platform.NetworkManager;
import net.jxta.protocol.DiscoveryResponseMsg;

import java.io.File;
import java.util.Enumeration;

/**
 * Illustrates the use of Discovery Service
 */
public class DiscoveryClient implements DiscoveryListener {

    private transient NetworkManager manager;
    private transient DiscoveryService discovery;

    /**
     * Constructor for the DiscoveryClient
     */
    public DiscoveryClient() {
        try {
            manager = new NetworkManager(NetworkManager.ConfigMode.EDGE,
               "DiscoveryClient", new File(new File(".cache"), "DiscoveryClient").toURI());
            manager.startNetwork();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }

        //Get the NetPeerGroup
        PeerGroup netPeerGroup = manager.getNetPeerGroup();
        // get the discovery service
        discovery = netPeerGroup.getDiscoveryService();
    }

    /**
     * main
     * @param args command line args
     */
    public static void main(String args[]) {
        DiscoveryClient disocveryClient = new DiscoveryClient();
        disocveryClient.start();
    }
```

```
/**
 * loop forever attempting to discover advertisements every minute
 */
public void start() {
    long waittime = 60 * 1000L;
    try {
        // Add ourselves as a DiscoveryListener for DiscoveryResponse events
        discovery.addDiscoveryListener(this);
        discovery.getRemoteAdvertisements(
                // no specific peer (propagate)
                null,
                //Adv type
                DiscoveryService.ADV,
                //Attribute = any
                null,
                //Value = any
                null,
                // one advertisement response is all we are looking for
                1,
                // no query specific listener. we are using a global listener
                null);
        while (true) {
            // wait a bit before sending a discovery message
            try {
                System.out.println("Sleeping for :" + waittime);
                Thread.sleep(waittime);
            } catch (Exception e) {
                // ignored
            }
            System.out.println("Sending a Discovery Message");
            // look for any peer
            discovery.getRemoteAdvertisements(
                    // no specific peer (propagate)
                    null,
                    //Adv type
                    DiscoveryService.ADV,
                    //Attribute = name
                    "Name",
                    //Value = the tutorial
                    "Discovery tutorial",
                    // one advertisement response is all we are looking for
                    1,
                    // no query specific listener. we are using a global listener
                    null);
        }
    } catch (Exception e) {
        e.printStackTrace();
    }
}

/**
 * This method is called whenever a discovery response is received, which are
 * either in response to a query we sent, or a remote publish by another node
 *
 * @param ev the discovery event
 */
public void discoveryEvent(DiscoveryEvent ev) {

    DiscoveryResponseMsg res = ev.getResponse();
    // let's get the responding peer's advertisement
    System.out.println(" [  Got a Discovery Response [" +
```

```
                res.getResponseCount() + " elements]  from peer : " + ev.getSource() + "  ]")

            Advertisement adv;
            Enumeration en = res.getAdvertisements();
            if (en != null) {
                while (en.hasMoreElements()) {
                    adv = (Advertisement) en.nextElement();
                    System.out.println(adv);
                }
            }
        }

        /**
         * Stops the platform
         */
        public void stop() {
            //Stop JXTA
            manager.stopNetwork();
        }
    }
```

Source Code: DiscoveryServer

```
package tutorial.discovery;

import net.jxta.discovery.DiscoveryEvent;
import net.jxta.discovery.DiscoveryListener;
import net.jxta.discovery.DiscoveryService;
import net.jxta.document.Advertisement;
import net.jxta.document.AdvertisementFactory;
import net.jxta.id.IDFactory;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupID;
import net.jxta.pipe.PipeService;
import net.jxta.platform.NetworkManager;
import net.jxta.protocol.DiscoveryResponseMsg;
import net.jxta.protocol.PipeAdvertisement;

import java.io.File;
import java.util.Enumeration;

/**
 * Illustrates the use of Discovery Service
 * Note this is for illustration purposes and is not meant as a blue-print
 */
public class DiscoveryServer implements DiscoveryListener {

    private transient NetworkManager manager;
    private transient DiscoveryService discovery;

    /**
     * Constructor for the DiscoveryServer
     */
    public DiscoveryServer() {
        try {
            manager = new NetworkManager(NetworkManager.ConfigMode.EDGE,
                  "DiscoveryServer", new File(new File(".cache"), "DiscoveryServer").toURI());
            manager.startNetwork();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }
        PeerGroup netPeerGroup = manager.getNetPeerGroup();
        // get the discovery service
        discovery = netPeerGroup.getDiscoveryService();
    }

    /**
     * main
     *
     * @param args command line args
     */
    public static void main(String args[]) {
        DiscoveryServer disocveryServer = new DiscoveryServer();
        disocveryServer.start();
    }

    /**
     * create a new pipe adv, publish it for 2 minut network time,
     * sleep for 3 minutes, then repeat
     */
    public void start() {
        long lifetime = 60 * 2 * 1000L;
        long expiration = 60 * 2 * 1000L;
        long waittime = 60 * 3 * 1000L;
```

```
        try {
            while (true) {
                PipeAdvertisement pipeAdv = getPipeAdvertisement();
                // publish the advertisement with a lifetime of 2 mintutes
                System.out.println("Publishing the following advertisement with lifetime :"
                                                        + lifetime + " expiration :" + expiration
                System.out.println(pipeAdv.toString());
                discovery.publish(pipeAdv, lifetime, expiration);
                discovery.remotePublish(pipeAdv, expiration);
                try {
                    System.out.println("Sleeping for :" + waittime);
                    Thread.sleep(waittime);
                } catch (Exception e) {
                    //ignored
                }
            }
        } catch (Exception e) {
            e.printStackTrace();
        }
    }

    /**
     * This method is called whenever a discovery response is received, which are
     * either in response to a query we sent, or a remote publish by another node
     *
     * @param ev the discovery event
     */
    public void discoveryEvent(DiscoveryEvent ev) {

        DiscoveryResponseMsg res = ev.getResponse();
        // let's get the responding peer's advertisement
        System.out.println(" [  Got a Discovery Response [" +
                res.getResponseCount() + " elements]  from peer : " + ev.getSource() + "  ]");

        Advertisement adv;
        Enumeration en = res.getAdvertisements();
        if (en != null) {
            while (en.hasMoreElements()) {
                adv = (Advertisement) en.nextElement();
                System.out.println(adv);
            }
        }
    }

    /**
     * Creates a pipe advertisement
     *
     * @return a Pipe Advertisement
     */
    public static PipeAdvertisement getPipeAdvertisement() {
        PipeAdvertisement advertisement = (PipeAdvertisement)
                AdvertisementFactory.newAdvertisement(PipeAdvertisement.getAdvertisementType()
        advertisement.setPipeID(IDFactory.newPipeID(PeerGroupID.defaultNetPeerGroupID));
        advertisement.setType(PipeService.UnicastType);
        advertisement.setName("Discovery tutorial");
        return advertisement;
    }

    /**
     * Stops the platform
     */
    public void stop() {
        //Stop JXTA
```

```
            manager.stopNetwork();
        }
    }
```

Chapter 9:Membership Service

This chapter introduces the Membership Service and provides examples of how the Membership Service is used.

Role of the Membership Service

The Membership Service is responsible for managing identities within JXTA. These identities, once established, are normally communicated as part of JXTA protocol messages. Identities allow applications and services to determine who is requesting an operation and decide whether that operation should be allowed. Applications may perform their own access control or may use the JXTA Access Service. The Access Service is a framework for making decisions regarding identities and operations.

Membership Service and Credentials

All Membership Services manage identities using a common structure, the JXTA Credential. A credential is the token which is passed between peers to communicate identity. Credentials are normally opaque, meaning that they can be interpreted only by the Membership service which created them. Credentials typically include validation mechanisms to ensure that they cannot be easily forged.
Credentials are created by Membership services as a result of an authentication process. Like the credentials themselves, the authentication process required to create a credential is very specific to the Membership service which uses it. It is normally difficult for applications and services to authenticate with a Membership service they were not written specifically for, however, the common JXTA Membership services all provide some mechanisms to allow for their plugability.

Local Credentials and Remote Credentials

The Membership service can generate two forms of credentials. Local credentials are generated via authentication and may be used by services and applications for messages to be sent by the local peer. Remote credentials are generated from messages received from remote peers and are used by applications and services to determine the identities of remote peers.

Membership Services and Peer Groups

The scope of a Membership service is a single peer group. In fact, each Membership service instance is associated with a peer group. The JXTA JSE implementation provides a Membership service with every peer group. Which Membership service is used for a particular peer group is part of the peer group's definition. A Membership service is started whenever the peer group is started. Since the Membership Service is a required service of JXTA JSE peer groups, all applications and services may depend on the fact that the Membership Service will always be available, though it may not always be possible to know what Membership Service a peer group may provide.
Applications and services normally interact with the Membership Service in two ways; via making API calls to the Membership service and by receiving callbacks from the Membership service or from Credentials issued by the Membership service.
The Membership service is started by the Peer Group along with the other peer group services as part of the `PeerGroup.init()` process. The Membership service will remain active as long as the peer group remains active. The Membership service is normally terminated when the peer group `stopApp()` method is called. In addition to shutting down the service, Membership service termination normally invalidates all of the issued credentials. The credential invalidation is normally only local—remote peers do not receive any notification that the credentials have been invalidated.

Membership Service and Other Services

Other Services may make use of the Membership service only after their own `startApp()` method is called. If a service requires the Membership Service as part of its operation, services should ensure its' availability in their own `startApp()` method using the following technique:

```
1
2    public int startApp(String args[]) {
3         Membership membership = group.getMembershipService();
4
5        if (null == membership) {
6            if (LOG.isEnabledFor(Level.WARN)) {
7                LOG.warn("Stalled until there is a membership service");
8            }
9            return Module.START_AGAIN_STALLED;
10        }
```

```
11
12          ...
```

The preceding code will cause the startApp() method to be called repeatedly by the `PeerGroup` until the Membership service is available or the Peer Group determines that the startup process is permanently stalled. Services should include similar checks for whichever other services they require as part of their `startApp()` method.

Membership Service Basic Operation

The most fundamental Membership service operation an application or service can use is the `getDefaultCredential()` method. This method returns the current default credential, assuming there is a current default. The credential returned can be safely considered as the most generic or relevant credential for the local peer within the current peer group. The credential will remain valid until the Membership Service is terminated or the credential expires or is otherwise invalidated. The application can check the validity of the credential using the `isValid()` method. The application or service may also provide a `CredentialListener` to the credential which will generate a callback when the credential becomes invalid or expires. This approach is often better than continually checking `isValid()`. In addition to `getDefaultCredential()`, applications can request the full set of active local credentials via `getCurrentCredentials()`. This will return all of the active local credentials. Applications can also register callback listeners to receive notifications whenever new local credentials are generated or the default credential changes.

Most applications and services, including the standard JXTA services, will use the default local credential in their protocol messages to identify the peer. It is generally not efficient to retrieve the default credential for each protocol message so most applications will use the callback interface in order to manage the credential they use to send with protocol messages.

The standard JXTA services use code similar to the following to manage the credentials they use in protocol messages. The code makes use of the callback listener to follow changes to the default credential. Since it is quite common to use the XML serialized version of credentials as part of JXTA protocols, the code also maintains a cached copy of the credential as an XML document.

> *Two important extensions that this code could use would be to check the validity of the credential before caching it and installing a listener for the cached credential becoming invalid.*

```
1   final static class  CurrentCredential {
2   /**
3    *     The current default credential
4    */
5    final Credential credential;
6
7   /**
8    *     The current default credential in serialized XML form.
9    */
10   final XMLDocument credentialDoc;
11
12    CurrentCredential(Credential credential, XMLDocument credentialDoc){
13       this.credential = credential;
14       this.credentialDoc = credentialDoc;
15       }
16    }
17}
18
19 /**
20  * The current Membership service default credential.
21  */
22  private CurrentCredential currentCredential;
23
24 /**
25  * Listener we use for membership property events.
26  */
27  private class CredentialListener implements PropertyChangeListener {
28
29    /**
30     *  {@inheritDoc}
31     */
32    public void propertyChange(PropertyChangeEvent evt) {
```

```
33        if (MembershipService.DEFAULT_CREDENTIAL_PROPERTY.equals(evt.getPropertyName())) {
34            if (LOG.isEnabledFor(Level.DEBUG)) {
35                LOG.debug("New default credential event");
36            }
37
38            synchronized (<Service>.this) {
39                Credential cred = (Credential) evt.getNewValue();
40                XMLDocument credentialDoc = null;
41
42                if (null != cred) {
43                    try {
44                        credentialDoc = cred.getDocument(MimeMediaType.XMLUTF8);
45
46                        currentCredential = new CurrentCredential(cred, credentialDoc);
47                    } catch (Exception all) {
48                        if (LOG.isEnabledFor(Level.WARN)) {
49                            LOG.warn("Could not generate credential document", all);
50                        }
51                    currentCredential = null;
52                    }
53                } else {
54                     currentCredential = null;
55                }
56           }
57        }
58    }
59}
60
61final CredentialListener membershipCredListener = new CredentialListener();
```

In the `startApp()` method the service registers with the callback listener and sets the initial credential to be used:

```
1public int startApp(String args[]) {
2
3...
4
5    synchronized (this) {
6        // register our credential listener.
7        membership.addPropertyChangeListener(MembershipService.DEFAULT_CREDENTIAL_PROPERTY,
membershipCredListener);
8
10            try {
11                // set the initial version of the default credential.
12                currentCredential = null;
13                Credential credential = membership.getDefaultCredential();
14                XMLDocument credentialDoc = null;
15
16                if (null != credential) {
17                    credentialDoc = credential.getDocument(MimeMediaType.XMLUTF8);
18                    currentCredential = new CurrentCredential(credential, credentialDoc);
19                }
20            } catch (Exception all) {
21                if (LOG.isEnabledFor(Level.WARN)) {
22                    LOG.warn("could not get default credential", all);
23                }
24            }
25        }
26
27...
28
29}
```

The `stopApp()` method includes code to de-register the listener and clear the cached default credential:

When the services wish to use the current credential as part of their protocol messages they use code similar to the following :

```
1 // Get the cached current credential. We establish a temporary
2 // variable since currentCredential may change at any time.
3 CurrentCredential activeCredential = currentCredential;
```

```
4
5 // Include the credential as part of our message.
6 if (null == activeCredential) {
7     protoMessage.setCredential(activeCredential.credentialDoc);
8 }
```

Authentication

Most Membership services do not provide a default credential upon startup. In order for the local peer to have established credentials and a default credential, an application must authenticate an identity. The first authenticated identity will also become the default credential. Services should not generally perform authentication but should use the default credential or credentials provided to them by the applications.
A Membership service may provide one or more authentication methods. Each authentication method generally provides a mechanism appropriate to a particular style of authentication. These can include programmatic interfaces, graphical user interfaces or other methods. Applications are free to chose the authentication method most appropriate for their needs. Every Membership service provides at least one authentication method and many provide an interactive GUI authenticator.
The Membership service authentication process has three distinct steps; setup, application and validate. The setup step occurs before the Membership service is invoked. Setup involves choosing an authentication method and can include, as appropriate, determining parameters for the authentication method and other details which a specific authentication method or Membership service may require. These parameters, and the choice of authentication method, are provided as an `AuthenticationCredential` to the Membership via the `apply()` method. If the Membership service determines that authentication can begin, then an Authenticator is returned.
The application step involves completing the Authenticator so that the `isReadyForJoin()` method returns true. The details of what is required in order to achieve a "`true`" result vary greatly between authenticators and between Membership services. It is not possible for an application to completely predict the requirements of an unfamiliar authenticator or Membership service.

```
1 MembershipService membership = group.getMembershipService();
2
3 AuthenticationCredential authCred =  new AuthenticationCredential(group,
                                                "StringAuthentication", null);
4
5  Authenticator auth = null;
6  try {
7     auth = (Authenticator) membership.apply(authCred);
8  } catch (Exception failed) {
9        ;
10 }
11
12 if (null != auth) {
13
14    ... authenticator is completed here ...
15
16    if (auth.isReadyForJoin()) {
17        membership.join(auth);
18    }
19 }
```

There are a couple of helpful conventions currently used by the common JXTA membership services and authenticators. All of the current JXTA authenticators provide a "`StringAuthentication`" authentication method which returns an Authenticator suitable for programmatic completion. The "`StringAuthentication`" authentication provides a number of methods which can be accessed via reflection to complete the parameters required for authentication. The authentication parameter set methods are all public methods who's names begin with "`setAuth`" and take a single "`String`" or "`char[]`" parameter. Following this prefix is a single digit indicating the order in which the parameters must (or possibly should) be provided to the authenticator. Following the single digit is an optional "_" (underscore) character. If present, this modifier indicates that the parameter is a password and should not be displayed in any user interface or otherwise shown. These password parameter methods also always provide an alternate method taking "`char[]`" as a parameter rather than a `String`.

This is done for security reasons, namely that when the Authenticator is finished using the char array it will clear it with zero characters to blank the password. The last part of the method name is the name of the parameter being set. The following are some examples from the PSE (Personal Security Environment) Membership service "StringAuthentication" Authenticator:

```
setAuth1_keystorePassword
setAuth2identity
setAuth3_identityPassword
```

You should not hardcode these method names into your program. The method names may change with future versions of the PSE StringAuthentication authenticator. You can find complete examples of how the "StringAuthentication" authenticator can be used as part of the JXTA Shell "login" command or as part of the myJXTA application.
In addition to the "StringAuthentication" method, the PSE Membership service also provides the "InteractiveAuthentication" method. This method provides a self contained minimal authentication graphical user interface. Membership services providing the "InteractiveAuthentication" method will return authenticators which can be cast to InteractiveAuthenticator and invoked via the interact() method. The interact() method returns a boolean result which indicates whether the user wishes to proceed with authentication.

```
1 Membership membership = group.getMembershipService();
3 AuthenticationCredential authCred = new AuthenticationCredential(group,
4                                         "InteractiveAuthentication", null);
5 InteractiveAuthenticator auth = (InteractiveAuthenticator) membership.apply(authCred);
6
7 if (auth.interact() && auth.isReadyForJoin()) {
8     membership.join(auth);
9 }
```

The current conventions for completing authenticators are only conventions. They are likely to be eventually replaced by more flexible and robust mechanisms such as the JAAS Authenticator framework. When this happens, the current authentication methods will likely be retained for a number of JXTA releases in order to allow time for applications to upgrade and convert to the new methods.
Once the Authenticator requirements have been satisfied, and isReadyForJoin() returns true for the final authentication step, validation can begin. At any time during the application step the authentication process may be abandoned by discarding the Authenticator object. The authentication validation step involves passing the completed Authenticator to the Membership Service's join() method. The join() method performs whatever validation is required upon the provided parameters and returns a Credential . The join() method may also generate errors if the provided parameters are incorrect or if their validity cannot be determined.
The Credential which results from the join() operation may be used by the application which requested it or provided to services for their use in protocols on the application's behalf.

Remote Credentials & Access Control

Remote Credentials are received from remote peers as part of protocol exchanges. Applications and services use the Membership service to transform the XML serialized credentials included in protocols into Credential objects. The following example extracts a remote credential from a message element, ensures that it is valid and checks to see if it is the credential for "somebody".

```
1 Credential remote = null;
2
3 try {
4    XMLDocument asDoc = (XMLDocument)
5        StructuredDocumentFactory.newStructuredDocument(messageElement);
6
7    Membership membership = group.getMembershipService();
8
9    remote = membership.makeCredential(asDoc);
10 } catch (Exception failed) {
11    if (LOG.isEnabledFor(Level.WARN)) {
12        LOG.warn("Could not ", failed);
13    }
14 }
15
16 if ((null != remote) && remote.isValid() &&
      “somebody”.equals(remote.getSubject())) {
17    // We have a valid credential
18   ...
19 }
```

The same credential could also be used with the JXTA Access service in order to determine whether some operation is supported. In the following example the remote credential is evaluated using the group's Access service to determine if the `ADD_USER` privileged operation is permitted. If it is permitted then the `performAddUser()` method is called with the protocol message.

```
1 Credential remote = null;
2
3 try {
4    XMLDocument asDoc = (XMLDocument)
5        StructuredDocumentFactory.newStructuredDocument(messageElement);
6
7    MembershipService membership = group.getMembershipService();
8    remote = membership.makeCredential(asDoc);
10 } catch (Exception failed) {
11    if (LOG.isEnabledFor(Level.WARN)) {
12        LOG.warn("Could not ", failed);
13    }
14 }
15
16 if (null != remote) {
17    // We have a credential
18   AccessService access = group.getAccessService();
19
20   boolean canDo = access.doAccessCheck(Operations.ADD_USER, remote);
21
22   if (canDo) {
23        performAddUser(message);
24   } else {
25        if (LOG.isEnabledFor(Level.WARN)) {
26             LOG.warn(Operations.ADD_USER.getSubject().toString() +
27                       “ not allowed  for “ +
28                       remote.getSubject().toString());
29        }
30   }
31 }
```

Chapter 10:PSE Membership Service

Introduction

This section covers usage of the PSE (Personal Security Environment) Membership service, a secure membership service which uses PKI (Public Key Infrastructure) certificates to implement Membership service functions. The PSE Membership service is the default Membership service for the Network Peer Group and is used by many other Peer Groups. This chapter will discuss how applications and services can utilize the PSE membership service of arbitrary Peer Groups and also how to plan for using the PSE Membership Service within user defined Peer Groups.

The PSE Membership Service and Other Services

There are a number of JXTA Services which are designed to work with the PSE Membership Service, these include the TLS (Transport Layer Security, see RFC 2246 : TLS v1.0) Message Transport and the CBJX (Crypto-based JXTA transfer) Message Transport. These two services specifically require the PSE Membership service to be the Membership service of the Peer Group in which they themselves are defined. The TLS and CBJX Message Transports provide different flavors of secured message transmission. TLS provides private, mutually authenticated, reliable streaming communications. CBJX provides lightweight secure message source verification.
The other JXTA Services (Resolver, Discovery, etc.) also make use of the PSE Membership service, when present, though their interactions are the same as would be for other Membership services.

The PSE Membership Service and Keystores

Every instance of the PSE Membership service is associated with a keystore. Keystores are used to store X.509 certificates and private keys. The certificates and certificate chains stored within a keystore are the certificates that have been trusted. In addition, the keystore also contains the certificates for the private keys. The private keys stored within a keystore can be used for signing messages.
Keystores can be provided by either software or hardware devices and virtually every keystore implementation provides some degree of key privacy and tamper resistance. The PSE Membership service supports a wide variety of keystore implementations through the KeyStoreManager interface including the default Java KeyStore (JKS) as well as PKCS#12 (the same keystore format used by Mozilla applications and Internet Explorer) and the PKCS#11 interface which allows JXTA to interface with JavaCard, SmartCard, TPM, iButton and other hardware keystores. The PSE Membership service can be used with a keystore that is initially empty or it can use a keystore that is already populated with certificates and private keys.

Accessing the PSE Membership Service

The PSE Membership service is defined in the package `net.jxta.impl.membership.pse.*` This is an implementation package. Normally applications are discouraged from accessing packages from the `net.jxta.impl.*` hierarchy because the implementation details are private and, unlike the API classes found in the `net.jxta.*` hierarchy, may change significantly from release to release. The PSE Membership service interface is, unlike other implementation classes, stable and applications may bind to it. In future JXTA releases a generic PKI Membership Service interface may be exported to the public JXTA API but for now applications may import the PSE Membership Service.

PSE Membership Service Components

The PSE Membership service consists of several components; the PSE Membership service class, the PSE Credential class, the PSE Config utility class and the PSE Authenticator classes. Applications are likely to use all four components. The PSE Membership service class implements the standard Membership service interface (`net.jxta.membership.Membership`) and also provides additional functionality for managing the PKI keystore. The PSE Credential class implements the standard Credential interface (`net.jxta.credential.Credential`) and provides additional functionality for accessing and using the underlying X.509 certificates. The PSE Config utility class is specific to the PSE Membership service and provides utility functions for manipulating the PSE keystore. The PSE Authenticators implement the standard Authenticator and InteractiveAuthenticator interfaces (`net.jxta.authentication.Authenticator` and `net.jxta.authentication.InteractiveAuthenticator`) and are normally used in the manner described in the previous chapter.

PSE Membership First Steps

The first step to take when using the PSE Membership service is determining if it is the active Membership service for

your peer group. This can be done via the following code:

```
1 MembershipService membership = peergroup.getMembershipService();
2 PSEMembershipService pseMembership = null;
3 ModuleImplAdvertisement implAdv =
4    (ModuleImplAdvertisement) membership.getImplAdvertisement();
5
6 if ((null != implAdv) &&
7    PSEMembershipService.pseMembershipSpecID.equals(
8       implAdv.getModuleSpecID()) &&
9     (membership instanceof PSEMembershipService)) {
10        pseMembership = (PSEMembershipService) membership;
11 }
```

The preceding code first acquires the active Membership service from the Peer Group and then recovers the Membership service's Module Implementation Advertisement. This advertisement is used by JXTA Service implementations to describe themselves in a standardized way. In some rare cases the Membership Service may not be able to provide the Module Implementation Advertisement so we must check for null. One of the fields the ModuleImplAdvertisement includes is the Module Specification ID. The Module Spec ID specifies the formal protocol interface which the service implements. In order to ensure that the Membership Service implements the correct protocol, so that our application may interact correctly with instances running on other peers, we check that the ModuleSpecID matches that of the PSEMembershipService. Finally, we check that the membership service is an instance of PSEMembershipService so that we can be sure that we can safely cast the membership service instance.
Once we have a PSE Membership service instance we can begin exploring it's functionality.

The PSE Credential

Every PSE Credential is based upon a chain of X.509 certificates. Local Credentials, those created by using the PSE Membership service to authenticate, also contain a private key (or reference to a private key in some cases). The PSE Credential allows applications and services to make use of the certificates and private key via extensions to the Credential API. The extension methods provide direct access to the certificate chain.

```
1 X509Certificate certificate = credential.getCertificate();
2 System.out.println(certificate.getSubject().toString());
3
4 X509Certificate certificates[] = credential.getCertificates();
5 for (x = 0; x < certificates.length; x++) {
6  System.out.print(certificates[x].getSubject().toString());
7  if (x < certificates.length ) {
8    System.out.print(“ is signed by “);
9  }
10
11 System.out.println();
12 }
```

Extension methods are also provided for accessing the private key. For some keystore implementations the private key is never exposed (it is kept in the hardware). Applications and services should avoid using the private key directly if possible to provide the greatest compatibility with keystore providers.
The primary operations which require credential and private key access are cryptographic signing and verification. Extension methods are provided for both of these operations. Here is an example of signing and verifying a block of data:

```
1 byte[] SignBytes(byte bytes[], PSECredential credential) {
2    Signature signer = credential.getSigner("SHA1WITHRSA");
3    signer.update(bytes);
4    return advSigner.sign();
5 }
6
7 boolean VerifyBytes(byte signature[], byte bytes[], PSECredential credential) {
8    Signature verifier = credential.getSignatureVerifier("SHA1WITHRSA");
9    verifier.update(bytes);
10   return verifier.verify(signature);
11 }
```

PSE Config

The PSE Membership service provides application and service access to the keystore via the

`PSEConfig` object. PSEConfig provides an interface similar to `java.security.KeyStore`. The PSEConfig interface supports two styles of operation. The first style requires that the keystore password be provided as a method parameter. The second style makes use of the keystore password which was set when the PSEConfig object was created or set via the `setKeyStorePassword()` method.

The methods which require a keystore password to be specified are provided for initializing (creating) the keystore and for managing the unlocking which takes place when the PSE Membership is first used. Other than for these purposes, the methods which make use of the preset keystore passphrase should be preferred.

The first step in using the PSEConfig interface is to determine if the PSE has been initialized and whether it has been unlocked. This can be completed using the following code:

```
1 PSEConfig pseConfig = pseMembership.getPSEConfig();
2 // Unlock the PSE using the keystore passphrase.
3 pseConfig.setKeyStorePassword(passphrase);
4
5 // Check if the PSE is initialized.
6 if (pseConfig.isInitialized()) {
7    KeyStore keystore = pseConfig.getKeyStore();
8
9     // Check if the PSE is unlocked.
10    if (null != keystore) {
11       println("Keystore is initialized and unlocked.");
12    }
13 }
```

If the PSE has not been previously initialized, the application can initialize the PSE via the following code:

```
1 PSEConfig pseConfig = pseMembership.getPSEConfig();
2
3 // Unlock the keystore with the user provided passphrase.
4 // Without this passphrase the result of isInitialized()
5 // may not be accurate.
6 pseConfig.setKeyStorePassword(passphrase);
7
8 // Check if the PSE is already initialized.
9  if (!pseConfig.isInitialized()) {
10    pseConfig.initialize();
11 }
```

There is an alternate mechanism for initializing the PSE. The configuration parameters for the PSE Membership service allow a seed certificate and private key to be specified. Once unlocked this certificate and private key will be used to initialize the PSE and will be added to the PSE. When the PSE has not been initialized, the PSE Membership behaves as though the only available key which may be authorized against is the seed private from the configuration.

Once initialized and unlocked, the PSE can be used to list, retrieve and add certificates and private keys. The operations available match those of the `java.security.KeyStore` interface fairly closely with one critical exception—the PSE allows only JXTA IDs as the aliases (names) for certificates and keys. It is not possible to access certificates or keys who's alias is not a valid JXTA ID. (It's important to note for KeyStoreManager providers that this is an API specific restriction. Within your implementation you may choose to map between aliases and IDs in any way you choose.)

The following code retrieves the list of the trusted certificates and for each determines if it is part of a public/private key pair, or just a trusted public key, and prints the subject:

```
1 PSEConfig pse = pseMembership.getPSEConfig();
2 Iterator<ID> eachCert = Arrays.asList(pse.getTrustedCertsList()).iterator();
3
4 while (eachCert.hasNext()) {
5    ID aCertID = eachCert.next();
6
7    boolean key = pse.isKey(aCertID);
8
9    X509Certificate cert = pse.getTrustedCertificate(aCert);
10
11   println(aCertID +  : " +
12        (key ? "Key Pair:" : "Trusted Cert:") +
13        cert.getSubjectX500Principal().getName());
14 }
```

The JXTA Shell contains a number of PSE specific shell commands. The shell commands provide good reference code for all of the common PSE operations that an application may wish to perform. You can find the source of the JXTA Shell PSE commands at: http://shell.jxta.org/source/browse/shell/binding/java/impl/src/net/jxta/impl/shell/bin/pse/

Chapter 11:Pipe Service

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:

- PipeServer — creates an input pipe with a precreated pipe ID, and listens for messages on this pipe
- PipeClient — creates an input pipe with a precreated pipe ID, creates an output pipe, and sends a message on this pipe

shows example output when the PipeListener application is run, and shows example output from the PipeServer application:

Example output: PipeServer.

```
Creating input pipe
Waiting for msgs on input pipe
Received message: Hello from peer suz-pipe[Wed Mar 26 16:27:15 PST 2007]
message received at: Wed Mar 26 16:27:16 PST 2003
```

Example output: PipeClient.

```
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
2007]
```

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 78

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 PipeServer and PipeClient 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.

PipeServer

This application creates and listens for messages on an input pipe. It defines a single class, PipeServer, 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
- `InputPipe pipeIn` — the input pipe that we create

`main()`

This method creates a new PipeServer object, calls NetworkManager.start() 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.)

`start()`

This method instantiates the JXTA platform and creates the default net peer group :

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

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

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

Next, we create a pipe advertisement by using a precreated pipe ID

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

```
pipeAdv = (PipeAdvertisement)
  AdvertisementFactory.newAdvertisement
        (MimeMediaType.XML_DEFAULTENCODING, 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.

`run()`

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

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

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

- `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 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 :

```
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 element with the specified namespace and name :

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

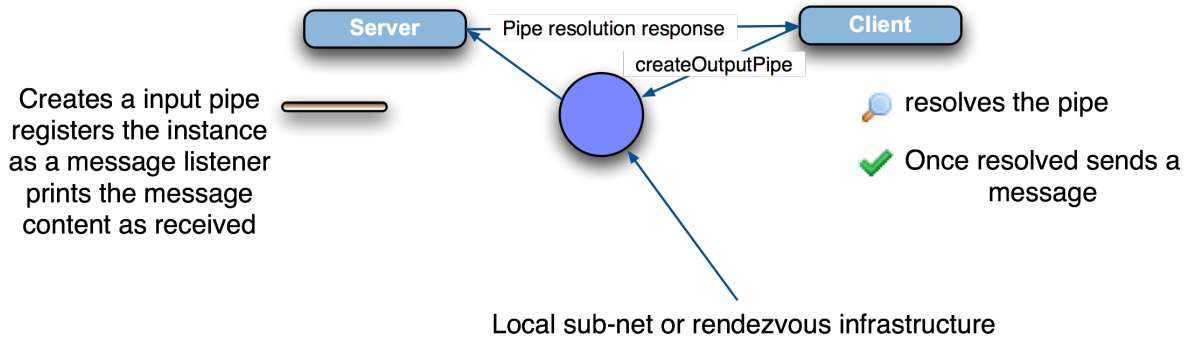
If an element with the specified namespace/name is not present within the message, this method returns null.

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 PipeServer class to refer to the message element name :

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

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

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



Source Code: PipeClient

```
package tutorial.pipe;

import net.jxta.document.AdvertisementFactory;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.StringMessageElement;
import net.jxta.id.IDFactory;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.OutputPipeEvent;
import net.jxta.pipe.OutputPipeListener;
import net.jxta.pipe.PipeID;
import net.jxta.pipe.PipeService;
import net.jxta.platform.NetworkManager;
import net.jxta.protocol.PipeAdvertisement;

import java.io.File;
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.Date;

/**
 * This tutorial illustrates the use of JXTA Pipes to exchange messages.
 * <p/>
 * This peer is the pipe "client". It opens the pipe for output and when it
 * resolves (finds a listening peer) it sends a message to the "server".
 */
public class PipeClient implements OutputPipeListener {

    /**
     * The tutorial message name space
     */
    public final static String MESSAGE_NAME_SPACE = "PipeTutorial";
    private boolean waitForRendezvous = false;
    private PipeService pipeService;
    private PipeAdvertisement pipeAdv;
    private OutputPipe outputPipe;
    private final Object lock = new String("lock");
    /**
     * Network is JXTA platform wrapper used to configure, start, and stop the
     * the JXTA platform
     */
    private NetworkManager manager;

    /**
     * A pre-baked PipeID string
     */
    public final static String PIPEIDSTR = "urn:jxta:uuid-
59616261646162614E50472050325033C0C1DE89719B456691A596B983BA0E1004";

    /**
     * Create this instance and starts the JXTA platform
     *
     * @param waitForRendezvous indicates whether to wait for a rendezvous connection
     */
    public PipeClient(boolean waitForRendezvous) {
        this.waitForRendezvous = waitForRendezvous;
        try {
            manager = new net.jxta.platform.NetworkManager(NetworkManager.ConfigMode.ADHOC,
                    "PipeClient", new File(new File(".cache"), "PipeClient").toURI());
            manager.startNetwork();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
```

```
        }
        // get the pipe service, and discovery
        pipeService = manager.getNetPeerGroup().getPipeService();
        // create the pipe advertisement
        pipeAdv = getPipeAdvertisement();
    }

    /**
     * main
     *
     * @param args command line arguments
     */
    public static void main(String args[]) {
        // by setting this property it will trigger a wait for a rendezvous
        // connection prior to attempting to resolve the pipe
        String value = System.getProperty("RDVWAIT", "false");
        boolean waitForRendezvous = Boolean.valueOf(value);
        PipeClient client = new PipeClient(waitForRendezvous);
        client.start();
    }

    /**
     * Creates the pipe advertisement
     * pipe ID
     *
     * @return the predefined Pipe Advertisement
     */
    public static PipeAdvertisement getPipeAdvertisement() {
        PipeID pipeID = null;
        try {
            pipeID = (PipeID) IDFactory.fromURI(new URI(PIPEIDSTR));
        } catch (URISyntaxException use) {
            use.printStackTrace();
        }
        PipeAdvertisement advertisement = (PipeAdvertisement)
                AdvertisementFactory.newAdvertisement(PipeAdvertisement.getAdvertisementType()
        advertisement.setPipeID(pipeID);
        advertisement.setType(PipeService.UnicastType);
        advertisement.setName("Pipe tutorial");
        return advertisement;
    }

    /**
     * the thread which creates (resolves) the output pipe
     * and sends a message once it's resolved
     */
    public synchronized void start() {
        try {
            if (waitForRendezvous) {
                System.out.println("Waiting for Rendezvous Connection");
                //wait indefinitely until connected to a rendezvous
                manager.waitForRendezvousConnection(0);
                System.out.println("Connected to Rendezvous, attempting to create a OutputPipe
            }
            // issue a pipe resolution asynchronously. outputPipeEvent() is called
            // once the pipe has resolved
            pipeService.createOutputPipe(pipeAdv, this);
            try {
                synchronized (lock) {
                    lock.wait();
                }
            } catch (InterruptedException e) {
                System.out.println("Thread interrupted");
            }
```

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


        /**
         * by implementing OutputPipeListener we must define this method which
         * is called when the output pipe is created
         *
         * @param event event object from which to get output pipe object
         */
        public void outputPipeEvent(OutputPipeEvent event) {

            System.out.println("Received the output pipe resolution event");
            // get the output pipe object
            outputPipe = event.getOutputPipe();

            Message msg;
            try {
                System.out.println("Sending message");
                //create the message
                msg = new Message();
                Date date = new Date(System.currentTimeMillis());
                //add a string message element with the current date
                StringMessageElement sme = new StringMessageElement(MESSAGE_NAME_SPACE,
    date.toString(), null);
                msg.addMessageElement(null, sme);
                //send the message
                outputPipe.send(msg);
                System.out.println("message sent");
            } catch (IOException e) {
                System.out.println("failed to send message");
                e.printStackTrace();
                System.exit(-1);
            }
            stop();
        }

        /**
         * Closes the output pipe and stops the platform
         */
        public void stop() {
            // Close the output pipe
            outputPipe.close();
            // Stop JXTA
            manager.stopNetwork();
            synchronized (lock) {
                // done.
                lock.notify();
            }
        }
    }
```

Source Code: PipeServer

```
    package tutorial.pipe;

    import net.jxta.document.MimeMediaType;
    import net.jxta.endpoint.Message;
    import net.jxta.endpoint.Message.ElementIterator;
    import net.jxta.endpoint.MessageElement;
    import net.jxta.endpoint.WireFormatMessage;
```

```
import net.jxta.endpoint.WireFormatMessageFactory;
import net.jxta.peergroup.PeerGroup;
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.PipeMsgEvent;
import net.jxta.pipe.PipeMsgListener;
import net.jxta.pipe.PipeService;
import net.jxta.platform.NetworkManager;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.util.CountingOutputStream;
import net.jxta.util.DevNullOutputStream;

import java.io.File;
import java.io.IOException;
import java.util.Date;

/**
 * This tutorial illustrates the use of JXTA Pipes to exchange messages.
 * <p/>
 * This peer is the pipe "server". It opens the pipe for input and waits for
 * messages to be sent. Whenever a Message is received from a "client" the
 * contents are printed.
 */
public class PipeServer implements PipeMsgListener {

    static PeerGroup netPeerGroup = null;

    /**
     * Network is JXTA platform wrapper used to configure, start, and stop the
     * the JXTA platform
     */
    transient NetworkManager manager;
    private PipeService pipeService;
    private PipeAdvertisement pipeAdv;
    private InputPipe inputPipe = null;

    /**
     * Constructor for the PipeServer object
     */
    public PipeServer() {
        manager = null;
        try {
            manager = new net.jxta.platform.NetworkManager(NetworkManager.ConfigMode.ADHOC,
                    "PipeServer", new File(new File(".cache"), "PipeServer").toURI());
            manager.startNetwork();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }

        //Get the NetPeerGroup
        netPeerGroup = manager.getNetPeerGroup();
        // get the pipe service, and discovery
        pipeService = netPeerGroup.getPipeService();
        // create the pipe advertisement
        pipeAdv = PipeClient.getPipeAdvertisement();
    }

    /**
     * main
     *
     * @param args command line args
     */
    public static void main(String args[]) {
        PipeServer server = new PipeServer();
```

```
        server.start();
    }

    /**
     * Dumps the message content to stdout
     *
     * @param msg     the message
     * @param verbose dumps message element content if true
     */
    public static void printMessageStats(Message msg, boolean verbose) {
        try {
            CountingOutputStream cnt;
            ElementIterator it = msg.getMessageElements();
            System.out.println("------------------Begin Message----------------------");
            WireFormatMessage serialed = WireFormatMessageFactory.toWire(
                    msg,
                    new MimeMediaType("application/x-jxta-msg"), null);
            System.out.println("Message Size :" + serialed.getByteLength());
            while (it.hasNext()) {
                MessageElement el = it.next();
                String eName = el.getElementName();
                cnt = new CountingOutputStream(new DevNullOutputStream());
                el.sendToStream(cnt);
                long size = cnt.getBytesWritten();
                System.out.println("Element " + eName + " : " + size);
                if (verbose) {
                    System.out.println("[" + el + "]");
                }
            }
            System.out.println("-------------------End Message----------------------");
        } catch (Exception e) {
            e.printStackTrace();

        }
    }

    /**
     * Creates the input pipe with this as the message listener
     */
    public void start() {

        try {
            System.out.println("Creating input pipe");
            // Create the InputPipe and register this for message arrival
            // notification call-back
            inputPipe = pipeService.createInputPipe(pipeAdv, this);
        } catch (IOException io) {
            io.printStackTrace();
            return;
        }
        if (inputPipe == null) {
            System.out.println(" cannot open InputPipe");
            System.exit(-1);
        }
        System.out.println("Waiting for msgs on input pipe");
    }

    /**
     * Closes the output pipe and stops the platform
     */
    public void stop() {
        // Close the input pipe
        inputPipe.close();
        //Stop JXTA
```

```
            manager.stopNetwork();
        }

        /**
         * PipeMsgListener interface for asynchronous message arrival notification
         *
         * @param event the message event
         */
        public void pipeMsgEvent(PipeMsgEvent event) {

            Message msg;
            try {
                // Obtain the message from the event
                msg = event.getMessage();
                if (msg == null) {
                    System.out.println("Received an empty message");
                    return;
                }
                // dump the message content to screen
                printMessageStats(msg, true);
            } catch (Exception e) {
                e.printStackTrace();
                return;
            }

            // get all the message elements
            Message.ElementIterator en = msg.getMessageElements();
            if (!en.hasNext()) {
                return;
            }

            // get the message element in the name space PipeClient.MESSAGE_NAME_SPACE
            MessageElement msgElement = msg.getMessageElement(null, PipeClient.MESSAGE_NAME_SPACE)
            // Get message
            if (msgElement.toString() == null) {
                System.out.println("null msg received");
            } else {
                Date date = new Date(System.currentTimeMillis());
                System.out.println("Message received at :" + date.toString());
                System.out.println("Message  created at :" + msgElement.toString());
            }
        }
    }
```

Chapter 12:Multicast Socket

Description

This example is intended as a porting guide to aide in the porting of a traditional use of java.net.MultiscastSocket to net.jxta.socket.JxtaMulticastSocket. A JxtaMulticastSocket is a sub-class of a java.net.MultiscastSocket, the difference being, is how it is bound. A java.net.MultiscastSocket is bound by a multicast address and a port number, where a JxtaMulticastSocket is bound by PeerGroup and PipeAdvertisement.

Learning goals

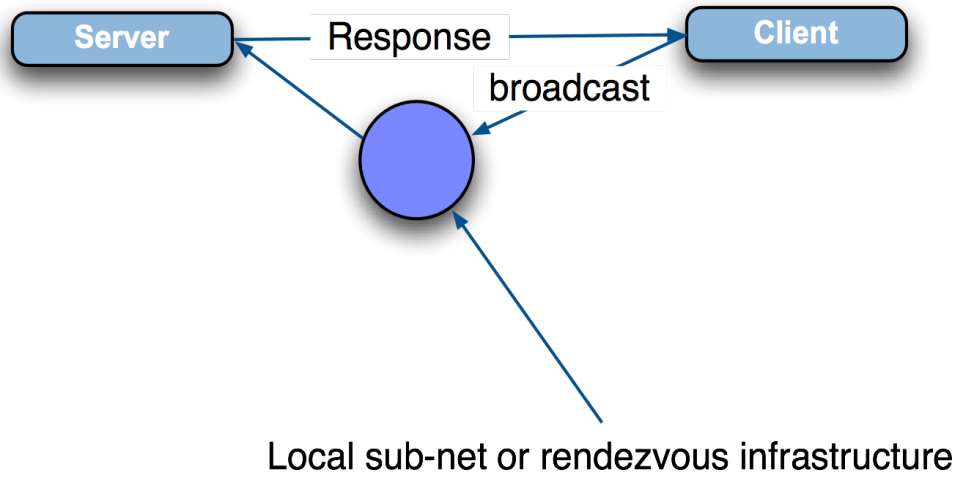
1. Construction and use of a JxtaMulticastSocket
2. Joining a virtual JXTA multicast group
3. Sending/receiving datagrams

Basic Operations

1. Construct a JxtaMulticastSocket with precreated pipe ID
2. Send/Receive a datagram
3. Stop the JXTA network

Caveats

1. A JxtaMulticastSocket operates over a propagated pipe, the scope of propagation is governed by the network topology, and the role of the node (Ad-Hoc, Edge, Rendezvous).
2. Datagram size is limited by the TcpTransport MulticastSocketSize, where multicast is utilized, and MTU size where a relay is utilized.
3. By default the example operates in Ad-Hoc mode. The configuration must be altered in order to operate differently.



Source Code : JxtaMulticastSocketClient

```
package tutorial.multicast;

import net.jxta.peergroup.PeerGroup;
import net.jxta.platform.NetworkManager;
import net.jxta.socket.JxtaMulticastSocket;

import java.io.File;
import java.io.IOException;
import java.net.DatagramPacket;
import java.util.Date;

/**
 * Simple example to illustrate the use of JxtaMulticastSocket
 */
public class JxtaMulticastSocketClient {

    /**
     * main
     *
     * @param args command line args
     */
    public static void main(String args[]) {
        NetworkManager manager = null;
        try {
            manager = new net.jxta.platform.NetworkManager(NetworkManager.ConfigMode.EDGE,
                      "JxtaMulticastSocketClient", new File(new File(".cache"),
                      "JxtaMulticastSocketClient").toURI());
            manager.startNetwork();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }

        PeerGroup netPeerGroup = manager.getNetPeerGroup();
        JxtaMulticastSocket mcastSocket = null;

        try {
            mcastSocket = new JxtaMulticastSocket(netPeerGroup,
                      JxtaMulticastSocketServer.getSocketAdvertisement());
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(-1);
        }

        Date date = new Date(System.currentTimeMillis());
        String hello = "Hello on : " + date.toString();
        try {
            DatagramPacket packet = new DatagramPacket(hello.getBytes(), hello.length());
            mcastSocket.send(packet);
            byte[] res = new byte[16384];
            DatagramPacket rpacket = new DatagramPacket(res, res.length);
            // It's likely we'll receive 2 packets a loopback and a response
            // loopback
            mcastSocket.receive(rpacket);
            // server response
            mcastSocket.receive(rpacket);
            String sw = new String(rpacket.getData(), 0, rpacket.getLength());
            System.out.println("Received data from :" + rpacket.getAddress());
            System.out.println(sw);
            //stop the platform
            manager.stopNetwork();
        } catch (IOException e) {
```

```
                e.printStackTrace();
            }
        }
    }
```

Source Code : JxtaMulticastSocketServer

```
package tutorial.multicast;

import net.jxta.document.AdvertisementFactory;
import net.jxta.id.IDFactory;
import net.jxta.peergroup.PeerGroup;
import net.jxta.pipe.PipeID;
import net.jxta.pipe.PipeService;
import net.jxta.platform.NetworkManager;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.socket.JxtaMulticastSocket;

import java.io.File;
import java.io.IOException;
import java.net.DatagramPacket;
import java.net.URI;
import java.net.URISyntaxException;

/**
 * Simple example to illustrate the use of JxtaMulticastSocket
 */

public class JxtaMulticastSocketServer {

    private static PeerGroup netPeerGroup = null;
    public final static String SOCKETIDSTR = "urn:jxta:uuid-
59616261646162614E5047205032503393B5C2F6CA7A41FDB0F890173088E79404";

    /**
     * Creates a Socket Pipe Advertisement with the predefined pipe ID
     *
     * @return a socket PipeAdvertisement
     */
    public static PipeAdvertisement getSocketAdvertisement() {
        PipeID socketID = null;
        try {
            socketID = (PipeID) IDFactory.fromURI(new URI(SOCKETIDSTR));
        } catch (URISyntaxException use) {
            use.printStackTrace();
        }
        PipeAdvertisement advertisement = (PipeAdvertisement)
                AdvertisementFactory.newAdvertisement(PipeAdvertisement.getAdvertisementType()
        advertisement.setPipeID(socketID);
        // set to type to propagate
        advertisement.setType(PipeService.PropagateType);
        advertisement.setName("Socket tutorial");
        return advertisement;
    }

    /**
     * main
     *
     * @param args command line args
     */
    public static void main(String args[]) {
        //Configures the JXTA platform
        net.jxta.platform.NetworkManager manager = null;
        try {
            manager = new net.jxta.platform.NetworkManager(NetworkManager.ConfigMode.ADHOC,
                    "JxtaMulticastSocketServer", new File(new File(".cache"),
"JxtaMulticastSocketServer").toURI());
            manager.startNetwork();
        } catch (Exception e) {
```

```
            e.printStackTrace();
            System.exit(-1);
        }

        System.out.println("Creating JxtaMulticastSocket");
        JxtaMulticastSocket mcastSocket = null;
        try {
            mcastSocket = new JxtaMulticastSocket(manager.getNetPeerGroup(),
getSocketAdvertisement());
            System.out.println("LocalAddress :" + mcastSocket.getLocalAddress());
            System.out.println("LocalSocketAddress :" + mcastSocket.getLocalSocketAddress());
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(-1);
        }

        byte[] buffer = new byte[16384];
        String ten4 = "Ten 4";

        DatagramPacket packet = new DatagramPacket(buffer, buffer.length);
        try {
            // wait for a datagram. The following can be put into a loop
            mcastSocket.receive(packet);
            String sw = new String(packet.getData(), 0, packet.getLength());
            System.out.println("Received data from :" + packet.getAddress());
            System.out.println(sw);
            //unicast back a response back to the remote
            DatagramPacket res = new DatagramPacket(ten4.getBytes(), ten4.length());
            res.setAddress(packet.getAddress());
            mcastSocket.send(res);
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Chapter 13:Propagated Pipe Example

PropagatedPipeClient

This example creates an output pipe and sends a message on it. It defines a single class, PipeClient, 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 TAG` — the message element name, or tag, which we will include in any message that we send

`main()`

This method creates a new PipeClient 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 :

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

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

- `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 :

```
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 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 :

```
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 :

```
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 PipeClient 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 :

```
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 :

```
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() :

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

After sending this message, we close the output pipe and return from this method :

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

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

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

After the pipe advertisement is created, the input stream is closed and the method returns:

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

Source Code: PropagatedPipeClient

```
package tutorial.propagated;

import net.jxta.endpoint.Message;
import net.jxta.endpoint.MessageElement;
import net.jxta.endpoint.StringMessageElement;
import net.jxta.endpoint.TextDocumentMessageElement;
import net.jxta.endpoint.MessageTransport;
import net.jxta.peergroup.PeerGroup;
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.PipeMsgEvent;
import net.jxta.pipe.PipeMsgListener;
import net.jxta.pipe.PipeService;
import net.jxta.platform.NetworkManager;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.protocol.RouteAdvertisement;
import net.jxta.impl.endpoint.router.EndpointRouter;
import net.jxta.impl.endpoint.router.RouteControl;
import net.jxta.document.XMLDocument;
import net.jxta.document.MimeMediaType;

import java.io.File;
import java.io.IOException;

/**
 * Simple example to illustrate the use of propagated pipes
 */
public class PropagatedPipeClient implements PipeMsgListener {
    private InputPipe inputPipe;
    private MessageElement routeAdvElement = null;
    private RouteControl routeControl = null;
    public static final String ROUTEADV = "ROUTE";

    /**
     * {@inheritDoc}
     */
    public void pipeMsgEvent(PipeMsgEvent event) {

        Message message = event.getMessage();
        if (message == null) {
            return;
        }
        MessageElement sel = message.getMessageElement(PropagatedPipeServer.NAMESPACE,
PropagatedPipeServer.PONGTAG);
        MessageElement nel = message.getMessageElement(PropagatedPipeServer.NAMESPACE,
PropagatedPipeServer.SRCNAMETAG);

        if (sel == null) {
            return;
        }
        //Since propagation relies on ip multicast whenever possible, it is to
        //to be expected that a unicasted message can be intercepted through ip
        //multicast
        System.out.println("Received a pong from :" + nel.toString() + " " + sel.toString());
    }


    /**
     * main
     *
     * @param args command line args
     */
    public static void main(String args[]) {
```

```
        PropagatedPipeClient client = new PropagatedPipeClient();
        NetworkManager manager = null;
        try {
            manager = new NetworkManager(NetworkManager.ConfigMode.ADHOC,
                    "PropagatedPipeClient", new File(new File(".cache"),
                    "PropagatedPipeClient").toURI());
            manager.startNetwork();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }
        PeerGroup netPeerGroup = manager.getNetPeerGroup();
        PipeAdvertisement pipeAdv = PropagatedPipeServer.getPipeAdvertisement();
        PipeService pipeService = netPeerGroup.getPipeService();
        System.out.println("Creating Propagated InputPipe for " + pipeAdv.getPipeID());
        try {
            client.inputPipe = pipeService.createInputPipe(pipeAdv, client);
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(-1);
        }
        MessageTransport endpointRouter =
(netPeerGroup.getEndpointService()).getMessageTransport("jxta");
        if (endpointRouter != null) {
            client.routeControl = (RouteControl)
endpointRouter.transportControl(EndpointRouter.GET_ROUTE_CONTROL, null);
            RouteAdvertisement route = client.routeControl.getMyLocalRoute();
            if (route != null) {
                client.routeAdvElement = new TextDocumentMessageElement(ROUTEADV,
                        (XMLDocument) route.getDocument(MimeMediaType.XMLUTF8), null);
            }
        }

        System.out.println("Creating Propagated OutputPipe for " + pipeAdv.getPipeID());
        OutputPipe output = null;
        try {
            output = pipeService.createOutputPipe(pipeAdv, 1);
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(-1);
        }
        int i = 0;
        try {
            while (i < 20) {
                Message ping = new Message();
                ping.addMessageElement(PropagatedPipeServer.NAMESPACE,
                        new StringMessageElement(PropagatedPipeServer.SRCIDTAG,
                                netPeerGroup.getPeerID().toString(),
                                null));
                ping.addMessageElement(PropagatedPipeServer.NAMESPACE,
                        new StringMessageElement(PropagatedPipeServer.SRCNAMETAG,
                                netPeerGroup.getPeerName() + " #" + i++,
                                null));
                if (client.routeAdvElement != null && client.routeControl != null) {
                    ping.addMessageElement(PropagatedPipeServer.NAMESPACE,
client.routeAdvElement);
                }

                System.out.println("Sending message :" + (i - 1));
                output.send(ping);
                Thread.sleep(500);
            }
            Thread.sleep(3000);
            manager.stopNetwork();
```

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

Source Code : PropagatedPipeServer

```
package tutorial.propagated;

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.XMLDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.MessageElement;
import net.jxta.endpoint.StringMessageElement;
import net.jxta.endpoint.MessageTransport;
import net.jxta.endpoint.TextDocumentMessageElement;
import net.jxta.id.IDFactory;
import net.jxta.peer.PeerID;
import net.jxta.peergroup.PeerGroup;
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.PipeID;
import net.jxta.pipe.PipeMsgEvent;
import net.jxta.pipe.PipeMsgListener;
import net.jxta.pipe.PipeService;
import net.jxta.platform.NetworkManager;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.protocol.RouteAdvertisement;
import net.jxta.impl.endpoint.router.RouteControl;
import net.jxta.impl.endpoint.router.EndpointRouter;

import java.io.File;
import java.io.IOException;
import java.net.URI;
import java.net.URISyntaxException;
import java.util.Collections;
import java.util.Map;
import java.util.Hashtable;
import java.util.logging.Level;

/**
 * Simple example to illustrate the use of propagated pipes
 */
public class PropagatedPipeServer implements PipeMsgListener {
    /**
     * Source PeerID
     */
    public final static String SRCIDTAG = "SRCID";
    /**
     * Source Peer Name
     */
    public final static String SRCNAMETAG = "SRCNAME";
    /**
     * Pong TAG name
     */
    public final static String PONGTAG = "PONG";
    /**
     * Tutorial message name space
     */
    public final static String NAMESPACE = "PROPTUT";
    private PeerGroup netPeerGroup = null;
    /**
     * Common propagated pipe id
     */
    public final static String PIPEIDSTR = "urn:jxta:uuid-
596162616461626l4E504720503250336FA944D18E8A4131AA74CE6F4BF85DEF04";
    private final static String completeLock = "completeLock";
```

```
    private static PipeAdvertisement pipeAdv = null;
    private static PipeService pipeService = null;
    InputPipe inputPipe = null;
    private transient Map<PeerID, OutputPipe> pipeCache = new Hashtable<PeerID, OutputPipe>();
    public static final String ROUTEADV = "ROUTE";
    private RouteControl routeControl = null;
    private MessageElement routeAdvElement = null;

    /**
     * Gets the pipeAdvertisement attribute of the PropagatedPipeServer class
     *
     * @return The pipeAdvertisement value
     */
    public static PipeAdvertisement getPipeAdvertisement() {
        PipeID pipeID = null;
        try {
            pipeID = (PipeID) IDFactory.fromURI(new URI(PIPEIDSTR));
        } catch (URISyntaxException use) {
            use.printStackTrace();
        }
        PipeAdvertisement advertisement = (PipeAdvertisement)
                AdvertisementFactory.newAdvertisement(PipeAdvertisement.getAdvertisementType()
        advertisement.setPipeID(pipeID);
        advertisement.setType(PipeService.PropagateType);
        advertisement.setName("Propagated Pipe Tutorial");
        return advertisement;
    }

    /**
     * {@inheritDoc}
     */
    public void pipeMsgEvent(PipeMsgEvent event) {

        Message message = event.getMessage();
        if (message == null) {
            return;
        }
        MessageElement sel = message.getMessageElement(NAMESPACE, SRCIDTAG);
        MessageElement nel = message.getMessageElement(NAMESPACE, SRCNAMETAG);
        // check for a route advertisement and train the endpoint router with the new route
        processRoute(message);
        if (sel == null) {
            return;
        }
        System.out.println("Received a Ping from :" + nel.toString());
        System.out.println("Source PeerID :" + sel.toString());
        Message pong = new Message();
        pong.addMessageElement(NAMESPACE,
                new StringMessageElement(PONGTAG, nel.toString(), null));
        pong.addMessageElement(NAMESPACE, new StringMessageElement(SRCNAMETAG,
netPeerGroup.getPeerName(), null));

        OutputPipe outputPipe = null;
        PeerID pid = null;
        try {
            pid = (PeerID) IDFactory.fromURI(new URI(sel.toString()));
            if (pid != null) {
                // Unicast the Message back. One should expect this to be unicast
                // in Rendezvous only propagation mode.
                // create a op pipe to the destination peer
                if (!pipeCache.containsKey(pid)) {
                    // Unicast datagram
                    // create a op pipe to the destination peer
                    outputPipe = pipeService.createOutputPipe(pipeAdv,
```

```
                                                    Collections.singleton(pid), 1);
                    pipeCache.put(pid, outputPipe);
                } else {
                    outputPipe = pipeCache.get(pid);
                }
                outputPipe.send(pong);
            } else {
                //send it to all
                System.out.println("unable to create a peerID from :" + sel.toString());
                outputPipe = pipeService.createOutputPipe(pipeAdv, 1000);
                outputPipe.send(pong);
            }
        } catch (IOException ex) {
            if (pid != null && outputPipe != null) {
                outputPipe.close();
                outputPipe = null;
                pipeCache.remove(pid);
            }
            ex.printStackTrace();
        } catch (URISyntaxException e) {
            e.printStackTrace();
        }
    }

    /**
     * Keep running, avoids existing
     */
    private void waitForever() {
        try {
            System.out.println("Waiting for Messages.");
            synchronized (completeLock) {
                completeLock.wait();
            }
            System.out.println("Done.");
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
    private void processRoute(final Message msg) {
        try {
            final MessageElement routeElement = msg.getMessageElement(NAMESPACE, ROUTEADV);
            if (routeElement != null && routeControl != null) {
                XMLDocument asDoc = (XMLDocument) StructuredDocumentFactory.
                newStructuredDocument(routeElement.getMimeType(), routeElement.getStream());
                final RouteAdvertisement route = (RouteAdvertisement)
                        AdvertisementFactory.newAdvertisement(asDoc);
                routeControl.addRoute(route);
            }
        } catch (IOException io) {
            io.printStackTrace();
        }
    }

    /**
     * main
     *
     * @param args command line args
     */
    public static void main(String args[]) {
        PropagatedPipeServer server = new PropagatedPipeServer();
        pipeAdv = getPipeAdvertisement();
        NetworkManager manager = null;
        try {
            manager = new NetworkManager(NetworkManager.ConfigMode.ADHOC,
```

```
                    "PropagatedPipeServer",
                    new File(new File(".cache"), "PropagatedPipeServer").toURI());
            manager.startNetwork();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }
        server.netPeerGroup = manager.getNetPeerGroup();
        pipeService = server.netPeerGroup.getPipeService();

        MessageTransport endpointRouter = (server.netPeerGroup.
                                getEndpointService()).getMessageTransport("jxta");
        if (endpointRouter != null) {
            server.routeControl = (RouteControl) endpointRouter.transportControl(
                                                    EndpointRouter.GET_ROUTE_CONTROL, null);
            RouteAdvertisement route = server.routeControl.getMyLocalRoute();
            if (route != null) {
                server.routeAdvElement = new TextDocumentMessageElement(ROUTEADV,
                        (XMLDocument) route.getDocument(MimeMediaType.XMLUTF8), null);
            }
        }

        System.out.println("Creating Propagated InputPipe for " + pipeAdv.getPipeID());
        try {
            server.inputPipe = pipeService.createInputPipe(pipeAdv, server);
        } catch (IOException e) {
            e.printStackTrace();
            System.exit(-1);
        }
        server.waitForever();
        server.inputPipe.close();
        manager.stopNetwork();
    }
}
```

Pipe Advertisement:

```
<!DOCTYPE jxta:PipeAdvertisement>
<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
 <Id>
  urn:jxta:uuid-
59616261646162614A757874614D504725184FBC4E5D498AA0919F662E40028B04
 </Id>
 <Type>
   JxtaPropagate
 </Type>
 <Name>
   PropagatedPipeClient/Server
 </Name>
</jxta:PipeAdvertisement>
```

Note – Both the `PropagatedPipeServer` and `PropagatedPipeClient` applications create the advertisement from a precreated PipeID.

Chapter 14:JxtaBiDiPipe Example

This example illustrates how to use the JxtaBiDiPipe to send messages between two JXTA peers. Two separate applications are used in this example:

- JxtaServerPipeExample —creates a JxtaServerPipe and awaits bi-directional connections
- JxtaBidiPipeExample — connects to a JxtaServerPipe and reliably exchanges messages over the connection,

shows example output when the JxtaServerPipeExample is run, and shows example input from the JxtaServerPipeExample:

Example output: JxtaServerPipeExample.

```
Reading in pipe.adv
Waiting for JxtaBidiPipe connections on JxtaServerPipe
JxtaBidiPipe accepted, sending 100 messages to the other end
Sending :Message #0
Sending :Message #1
Sending :Message #2
Sending :Message #3
Sending :Message #4
Sending :Message #5
Sending :Message #6
Sending :Message #7
Sending :Message #8
Sending :Message #9
Sending :Message #10
```

Example output: JxtaBidiPipeExample.

```
reading in pipe.adv
creating the BiDi pipe
Attempting to establish a connection
Message  :Message #0
Message  :Message #1
Message  :Message #2
Message  :Message #3
Message  :Message #4
Message  :Message #5
Message  :Message #6
Message  :Message #7
Message  :Message #8
Message  :Message #9
Message  :Message #10
```

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 JxtaServerPipeExample application must be run first. After the pipe advertisement is created JxtaServerPipeExample listens for JxtaBiDiPipe connections.

JxtaBiDiPipe

The JxtaBiDiPipe uses the core JXTA uni-directional pipes (InputPipe and OutputPipe) to simulate bi-directional pipes in the platform J2SE binding. The JxtaServerPipe defines the following methods:

- bind — binds to the pipe within the specified group
- connect — connects JxtaBiDiPipe to a JxtaServerPipe within the specified group
- setPipeTimeout — Sets the Timeout to establish a JxtaBiDiPipe connection

JxtaBiDiPipe defines the following methods:

- setReliable() — toggles reliability
- setListener() — registers a message listener for asynchronous message delivery

- sendMessage — asynchronously delivers a message
- getMessage() — synchronously waits for messages within specified timeout
- setPipeTimeout — Sets the Timeout to establish a JxtaBiDiPipe connection

JxtaServerPipeExample

This application creates JxtaServerPipe and awaits connections. File pipe.adv contains the pipe advertisement which both ends bind to.

- `String SenderMessage` — 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
- `JxtaServerPipe serverPipe` — the JxtaServerPipe we use to accept connections and receive messages

main()

This method creates a new JxtaServerPipeExample object, calls startJxta() to instantiate the JXTA platform and create the default net peer group, and calls run().

run()

This method uses the JxtaServerPipe to accept connections, and send messages.

```
JxtaBiDiPipe bipipe = serverPipe.accept();
```

Once a connection is returned, JxtaServerPipeExample send 100, then resumes to accept new connections. This step is repeated until the process is killed.

Source Code: JxtaServerPipeExample

```
package tutorial.bidi;

import net.jxta.document.AdvertisementFactory;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.MessageElement;
import net.jxta.endpoint.StringMessageElement;
import net.jxta.id.ID;
import net.jxta.logging.Logging;
import net.jxta.peergroup.PeerGroup;
import net.jxta.pipe.PipeID;
import net.jxta.pipe.PipeMsgEvent;
import net.jxta.pipe.PipeMsgListener;
import net.jxta.pipe.PipeService;
import net.jxta.platform.NetworkManager;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.util.JxtaBiDiPipe;
import net.jxta.util.JxtaServerPipe;

import java.io.File;
import java.net.URI;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * This is the Server side of the example. This example awaits bidirectioncal
 * pipe connections, then spawn a new thread to deal with The connection
 */
public class JxtaServerPipeExample {
    /**
     * Number of messages to send
     */
    public final static int ITERATIONS = 10000;
    private transient PeerGroup netPeerGroup = null;
    private transient JxtaServerPipe serverPipe;
    private final static Logger LOG = Logger.getLogger(JxtaServerPipeExample.class.getName());
    private final static String SenderMessage = "pipe_tutorial";
    private final static String PIPEIDSTR = "urn:jxta:uuid-
59616261646162614E50472050325033894BCED387C4A2BBD8E9411B78C284104";
    private transient NetworkManager manager = null;
    private final transient File home = new File(new File(".cache"), "server");
    private final String receipt = "Receipt";

    /**
     * Constructor for the JxtaServerPipeExample object
     */
    public JxtaServerPipeExample() {
        try {
            manager = new NetworkManager(NetworkManager.ConfigMode.ADHOC,
                                         "JxtaServerPipeExample", home.toURI());
            manager.startNetwork();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }
        netPeerGroup = manager.getNetPeerGroup();
    }

    /**
     * Gets the pipeAdvertisement attribute of the JxtaServerPipeExample class
     *
     * @return The pipeAdvertisement
     */
    public static PipeAdvertisement getPipeAdvertisement() {
```

```
        PipeID pipeID = (PipeID) ID.create(URI.create(PIPEIDSTR));
        PipeAdvertisement advertisement = (PipeAdvertisement)
                AdvertisementFactory.newAdvertisement(PipeAdvertisement.getAdvertisementType()
        advertisement.setPipeID(pipeID);
        advertisement.setType(PipeService.UnicastType);
        advertisement.setName("JxtaBiDiPipe tutorial");
        return advertisement;
    }

    /**
     * Connection wrapper. Once started, it send a message, awaits a response.
     * repeats the above steps for a predefined number of iterations
     */
    private class ConnectionHandler implements Runnable, PipeMsgListener {
        JxtaBiDiPipe pipe = null;

        /**
         * Constructor for the ConnectionHandler object
         *
         * @param pipe message pipe
         */
        ConnectionHandler(JxtaBiDiPipe pipe) {
            this.pipe = pipe;
            pipe.setMessageListener(this);
        }

        /**
         * {@inheritDoc}
         */
        public void pipeMsgEvent(PipeMsgEvent event) {
            Message msg;
            synchronized (receipt) {
                // for every message we get, we pong
                receipt.notify();
            }
            try {
                // grab the message from the event
                msg = event.getMessage();
                if (msg == null) {
                    if (Logging.SHOW_FINE && LOG.isLoggable(Level.FINE)) {
                        LOG.fine("Received an empty message, returning");
                    }
                    return;
                }
                if (Logging.SHOW_FINE && LOG.isLoggable(Level.FINE)) {
                    LOG.fine("Received a response");
                }
                // get the message element named SenderMessage
                MessageElement msgElement = msg.getMessageElement(SenderMessage, SenderMessage
                // Get message
                if (msgElement.toString() == null) {
                    System.out.println("null msg received");
                } else {
                    //System.out.println("Got Message :" + msgElement.toString());
                }
            } catch (Exception e) {
                if (Logging.SHOW_FINE && LOG.isLoggable(Level.FINE)) {
                    LOG.fine(e.toString());
                }
            }
        }

        /**
         * Send a series of messages over a pipe
```

```
     *
     * @param pipe Description of the Parameter
     */
    private void sendTestMessages(JxtaBiDiPipe pipe) {
        long t0, t1, t2, t3, delta;
        try {
            t2 = System.currentTimeMillis();
            for (int i = 0; i < ITERATIONS; i++) {
                t0 = System.currentTimeMillis();
                Message msg = new Message();
                String data = "Seq #" + i;
                msg.addMessageElement(SenderMessage,
                        new StringMessageElement(SenderMessage,
                                data,
                                null));
                //System.out.print("Sending :" + data);
                //t0 = System.currentTimeMillis();
                pipe.sendMessage(msg);
                synchronized (receipt) {
                    receipt.wait(1000);
                }
                t1 = System.currentTimeMillis();
                delta = (t1 - t0);
                if (delta > 50) {
                    System.out.println(" completed message sequence #" + i + " in :"
                                                + delta);
                }
            }
            t3 = System.currentTimeMillis();
            System.out.println(" completed " + ITERATIONS / ((t3 - t2) / 1000)
                                        + " transactions/sec. Total time :" + (t3 - t2));
        } catch (Exception ie) {
            ie.printStackTrace();
        }
    }

    /**
     * Main processing method for the ConnectionHandler object
     */
    public void run() {
        try {
            sendTestMessages(pipe);
            pipe.close();
        } catch (Exception e) {
            e.printStackTrace();
        }
    }
}

/**
 * Main processing method for the JxtaServerPipeExample object
 */
public void run() {

    System.out.println("Waiting for JxtaBidiPipe connections on JxtaServerPipe");
    while (true) {
        try {
            JxtaBiDiPipe bipipe = serverPipe.accept();
            if (bipipe != null) {
                System.out.println("JxtaBidiPipe accepted, sending "
                                            + ITERATIONS + " messages to the other end");
                //Send a 100 messages
                Thread thread = new Thread(new ConnectionHandler(bipipe),
                                                        "Connection Handler Thread");
```

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

    /**
     * main
     *
     * @param args command line args
     */
    public static void main(String args[]) {

        JxtaServerPipeExample eg = new JxtaServerPipeExample();
        try {
            System.out.println(JxtaServerPipeExample.getPipeAdvertisement().toString());
            eg.serverPipe = new JxtaServerPipe(eg.netPeerGroup,
                                               JxtaServerPipeExample.getPipeAdvertisement());
            //block forever until a connection is accepted
            eg.serverPipe.setPipeTimeout(0);
        } catch (Exception e) {

            System.out.println("failed to bind to the JxtaServerPipe due to
                                                        the following exception");
            e.printStackTrace();
            System.exit(-1);
        }
        // run on this thread
        eg.run();
    }
}
```

An example pipe advertisement, is listed below:

```
<!DOCTYPE jxta:PipeAdvertisement>

<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
        <Id>
urn:jxta:uuid-59616261646162614E50472050325033894 4BCED387C4A2BBD8E9415B78C484104
        </Id>
        <Type>
                JxtaUnicast
        </Type>
        <Name>
                ServerPipe tutorial
        </Name>
</jxta:PipeAdvertisement>
```

JxtaBidiPipeExample

This application creates JxtaDiDiPipe and attempts to connect to JxtaServerPipe.

- `String SenderMessage` — the message element name, or tag, which we must include in any message we send to the JxtaServerPipeExample (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
- `JxtaBiDiPipe pipe` — the JxtaBiDiPipe used to connect to JxtaServerPipe

main()

This method creates a new JxtaBidiPipeExample object, calls startJxta() to instantiate the JXTA platform and create the default net peer group, and then awaits to be notified when messages arrive. When a messages the message content is printed on the console.

```
System.out.println("creating the BiDi pipe");
eg.pipe = new JxtaBiDiPipe();
// ensure reliability
eg.pipe.setReliable(true);
System.out.println("Attempting to establish a connection")
eg.pipe.connect(eg.netPeerGroup,
                // any peer listening on the server pipe will do
                null,
                eg.pipeAdv,
                // wait upto 3 minutes
                180000,
                // register as a message listener
                eg);
```

Source Code: JxtaBidiPipeExample

```
package tutorial.bidi;

import net.jxta.endpoint.Message;
import net.jxta.endpoint.MessageElement;
import net.jxta.logging.Logging;
import net.jxta.peergroup.PeerGroup;
import net.jxta.pipe.PipeMsgEvent;
import net.jxta.pipe.PipeMsgListener;
import net.jxta.platform.NetworkManager;
import net.jxta.util.JxtaBiDiPipe;

import java.io.File;
import java.io.IOException;
import java.util.logging.Level;
import java.util.logging.Logger;

/**
 * This example illustrates how to utilize a JxtaBiDiPipe to establish a
 * bidirectional connection, and also illustrates use of asynchronous messaging
 * interface.
 * The example will attempt to establish a connection to it's counterpart
 * (JxtaServerPipeExample), then wait until all the messages are received
 * asynchronously
 */

public class JxtaBidiPipeExample implements PipeMsgListener {
    private final static Logger LOG = Logger.getLogger(JxtaBidiPipeExample.class.getName());
    private transient NetworkManager manager = null;
    private final transient File home = new File(new File(".cache"), "client");
    private transient JxtaBiDiPipe pipe;
    private final static String SenderMessage = "pipe_tutorial";
    private final static String completeLock = "completeLock";
    private int count = 0;

    public JxtaBidiPipeExample(boolean waitForRendezvous) {
        try {
            manager = new NetworkManager(NetworkManager.ConfigMode.ADHOC,
                                         "JxtaBidiPipeExample", home.toURI());
            manager.startNetwork();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }
        //manager.login("principal", "password");
        PeerGroup netPeerGroup = manager.getNetPeerGroup();
        try {
            pipe = new JxtaBiDiPipe();
            pipe.setReliable(true);
            if (waitForRendezvous) {
                // wait until a connection to a rendezvous is established
                manager.waitForRendezvousConnection(0);
            }
            System.out.println("Attempting to establish a connection");
            pipe.connect(netPeerGroup,
                    null,
                    JxtaServerPipeExample.getPipeAdvertisement(),
                    60000,
                    // register as a message listener
                    this);
            //at this point we need to keep references around until data exchange
            //is complete
            System.out.println("JxtaBiDiPipe pipe created");
            waitUntilCompleted();
```

```
            manager.stopNetwork();
        } catch (IOException e) {
            System.out.println("failed to bind the JxtaBiDiPipe due to the following exceptio
            e.printStackTrace();
            System.exit(-1);
        }
    }

    /**
     * This is the PipeListener interface. Expect a call to this method
     * When a message is received.
     * when we get a message, print out the message on the console
     *
     * @param event message event
     */
    public void pipeMsgEvent(PipeMsgEvent event) {

        Message msg;
        Message response;
        try {
            // grab the message from the event
            msg = event.getMessage();
            if (msg == null) {
                if (Logging.SHOW_FINE && LOG.isLoggable(Level.FINE)) {
                    LOG.fine("Received an empty message, returning");
                }
                return;
            }
            if (Logging.SHOW_FINE && LOG.isLoggable(Level.FINE)) {
                LOG.fine("Received a response");
            }
            // get the message element named SenderMessage
            MessageElement msgElement = msg.getMessageElement(SenderMessage, SenderMessage);
            // Get message
            if (msgElement.toString() == null) {
                System.out.println("null msg received");
            } else {
                //System.out.println("Got Message  :"+ msgElement.toString());
                count++;
            }
            response = msg.clone();
            //System.out.println("Sending response to " + msgElement.toString());
            pipe.sendMessage(response);
            // If JxtaServerPipeExample.ITERATIONS # of messages received, it is
            // no longer needed to wait. notify main to exit gracefully
            if (count >= JxtaServerPipeExample.ITERATIONS) {
                synchronized (completeLock) {
                    completeLock.notify();
                }
            }
        } catch (Exception e) {
            if (Logging.SHOW_FINE && LOG.isLoggable(Level.FINE)) {
                LOG.fine(e.toString());
            }
        }
    }

    private void waitUntilCompleted() {
        try {
            System.out.println("Waiting for Messages.");
            synchronized (completeLock) {
                completeLock.wait();
            }
            pipe.close();
```

```
                System.out.println("Done.");
            } catch (Exception e) {
                e.printStackTrace();
            }
        }

        /**
         * main
         *
         * @param args command line args
         */
        public static void main(String args[]) {
            String value = System.getProperty("RDVWAIT", "false");
            boolean waitForRendezvous = Boolean.valueOf(value);
            new JxtaBidiPipeExample(waitForRendezvous);
        }
    }
```

Chapter 15:JxtaSocket Tutorial

JxtaSocket is a bidirectional Pipe, which implements the java.net.Socket interface. JxtaSockets behave as closely as
possible as a regular java socket, with the following differences:

- JxtaSockets does not implement Nagle's algorithm and therefore applications must flush data at the end of data
  transmission, or as the application necessitates.
- JxtaSockets does not implement keep alive which, for most applications it is not required, however is good to note
  this difference.

This example illustrates how to use the JxtaSockets to send data between two JXTA peers. Two separate applications
are used in this example:

- JxtaServerSocketExample —creates a JxtaServerSocket and awaits bi-directional connections
- JxtaSocketExample — connects to a JxtaSocket and reliably exchanges data over the connection,

shows example output when the JxtaServerPipeExample is run, and shows example input from the JxtaServerSocketExample:

Example output: JxtaSocketExample.

```
Starting JXTA
reading in socket.adv
Connecting to the server
Reading in data
received 299 bytes
Sending back 65536 * 1824 bytes
Completed in :21673 msec
Data Rate :43089 Kbit/sec
Connecting to the server
Reading in data
received 299 bytes
Sending back 65536 * 1824 bytes
Completed in :14743 msec
Data Rate :63344 Kbit/sec
```

Example output: JxtaServerSocketExample.

```
Reading in socket.adv
starting ServerSocket
Calling accept
socket created
299 bytes sent
```

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 JxtaServerSocketExample application must be run first. It reads in pipe.adv which contains the pipe advertisement. After the pipe advertisement is read JxtaServerSocketExample listens for JxtaSocket connections.

JxtaSocket

The JxtaSocket uses the core JXTA uni-directional pipes (InputPipe and OutputPipe) to simulate a socket connection in the platform J2SE binding.

The JxtaServerSocket defines the following methods:

- bind — binds to the pipe within the specified group
- accept — waits for JxtaSocket connections within the specified group
- setSoTimeout — Sets the ServerSocket Timeout

JxtaSocket defines the following methods:

- create() — toggles reliability
- getOutputStream ()— returns the output stream for the socket
- getInputStream() — returns the intput stream for the socket
- setSoTimeout() — Sets the Socket Timeout

JxtaServerSocketExample

We define the following instance fields:

- `PeerGroup netPeerGroup` — our peer group, the default net peer group
- `JxtaServerSocket serverSocket` — the JxtaServerSocket we use to accept connections

`main()`

This method creates a new JxtaServerPipeExample object, calls startJxta() to instantiate the JXTA platform and create the default net peer group, and calls run().

`construc tor()`

This method instantiates the JXTA platform and creates the default net peer group :

```
manager = new NetworkManager("SocketServer", home);
```

`run()`

This method uses the JxtaServerSocket to accept connections, and send and receive data:

```
Socket socket = serverSocket.accept();
```

Once a connection is returned, JxtaServerSocketExample sends randomly generated data, while receiving data from the remote side

Source Code: SocketServer

```
package tutorial.socket;
import net.jxta.document.AdvertisementFactory;
import net.jxta.exception.PeerGroupException;
import net.jxta.id.IDFactory;
import net.jxta.peergroup.PeerGroup;
import net.jxta.pipe.PipeID;
import net.jxta.pipe.PipeService;
import net.jxta.platform.NetworkManager;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.socket.JxtaServerSocket;

import java.io.DataInput;
import java.io.DataInputStream;
import java.io.File;
import java.io.IOException;
import java.io.InputStream;
import java.io.OutputStream;
import java.net.Socket;
import java.net.URI;
import java.net.URISyntaxException;
import java.text.MessageFormat;

/**
 * This tutorial illustrates the use JxtaServerSocket It creates a
 * JxtaServerSocket with a back log of 10. it also blocks indefinitely, until a
 * connection is established.
 * <p/>
 * Once a connection is established data is exchanged with the initiator.
 * The initiator will provide an iteration count and buffer size. The peers will
 * then read and write buffers. (or write and read for the initiator).
 */
public class SocketServer {

    private transient PeerGroup netPeerGroup = null;
    public final static String SOCKETIDSTR = "urn:jxta:uuid-
59616261646162614E5047205032503393B5C2F6CA7A41FBB0F890173088E79404";

    public SocketServer() throws IOException, PeerGroupException {
        NetworkManager manager = new NetworkManager(NetworkManager.ConfigMode.ADHOC,
                                                    "SocketServer",
                                                    new File(new File(".cache"),
"SocketServer").toURI());
        manager.startNetwork();
        netPeerGroup = manager.getNetPeerGroup();
    }

    public static PipeAdvertisement createSocketAdvertisement() {
        PipeID socketID = null;
        try {
            socketID = (PipeID) IDFactory.fromURI(new URI(SOCKETIDSTR));
        } catch (URISyntaxException use) {
            use.printStackTrace();
        }
        PipeAdvertisement advertisement = (PipeAdvertisement)
                AdvertisementFactory.newAdvertisement(PipeAdvertisement.getAdvertisementType(
        advertisement.setPipeID(socketID);
        advertisement.setType(PipeService.UnicastType);
        advertisement.setName("Socket tutorial");
        return advertisement;
    }

    /**
```

```
     * Wait for connections
     */
    public void run() {

        System.out.println("Starting ServerSocket");
        JxtaServerSocket serverSocket = null;

        try {
            serverSocket = new JxtaServerSocket(netPeerGroup, createSocketAdvertisement(), 10
            serverSocket.setSoTimeout(0);
        } catch (IOException e) {
            System.out.println("failed to create a server socket");
            e.printStackTrace();
            System.exit(-1);
        }

        while (true) {
            try {
                System.out.println("Waiting for connections");
                Socket socket = serverSocket.accept();
                if (socket != null) {
                    System.out.println("New socket connection accepted");
                    Thread thread = new Thread(new ConnectionHandler(socket),
                                                        "Connection Handler Thread");
                    thread.start();
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    private class ConnectionHandler implements Runnable {
        Socket socket = null;

        ConnectionHandler(Socket socket) {
            this.socket = socket;
        }

        /**
         * Sends data over socket
         *
         * @param socket the socket
         */
        private void sendAndReceiveData(Socket socket) {
            try {
                long start = System.currentTimeMillis();

                // get the socket output stream
                OutputStream out = socket.getOutputStream();
                // get the socket input stream
                InputStream in = socket.getInputStream();
                DataInput dis = new DataInputStream(in);

                long iterations = dis.readLong();
                int size = dis.readInt();
                long total = iterations * size * 2L;
                long current = 0;

                System.out.println(MessageFormat.format("Sending/Receiving {0} bytes.", total)
                while (current < iterations) {
                    byte[] buf = new byte[size];
                    dis.readFully(buf);
                    out.write(buf);
```

```
                    out.flush();
                    current++;
                }

                out.close();
                in.close();

                long finish = System.currentTimeMillis();
                long elapsed = finish - start;

                System.out.println(MessageFormat.format("EOT. Received {0} bytes in {1} ms.
                                   Throughput = {2} KB/sec.",
                                   total, elapsed, (total / elapsed) * 1000 / 1024));
                socket.close();
                System.out.println("Connection closed");
            } catch (Exception ie) {
                ie.printStackTrace();
            }
        }

        public void run() {
            sendAndReceiveData(socket);
        }
    }

    /**
     * main
     *
     * @param args command line args
     */
    public static void main(String args[]) {
        /*
        System.setProperty("net.jxta.logging.Logging", "FINEST");
        System.setProperty("net.jxta.level", "FINEST");
        System.setProperty("java.util.logging.config.file", "logging.properties");
        */
        try {
            Thread.currentThread().setName(SocketServer.class.getName() + ".main()");
            SocketServer socEx = new SocketServer();
            socEx.run();
        } catch (Throwable e) {
            System.err.println("Failed : " + e);
            e.printStackTrace(System.err);
            System.exit(-1);
        }
    }
}
```

Pipe adertisement example

An example pipe advertisement, is listed below:

```
<!DOCTYPE jxta:PipeAdvertisement>

<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
        <Id>
urn:jxta:uuid-59616261646162614E504720503250339 3B5C2F6CA7A41FBB0F890173088E79404
        </Id>
        <Type>
                JxtaUnicast
        </Type>
        <Name>
                socket tutorial
        </Name>
</jxta:PipeAdvertisement>
```

ClientSocket

This application creates JxtaSocket and attempts to connect to JxtaServerSocket.

- We define the following instance fields:
- `PeerGroup netPeerGroup` — our peer group, the default net peer group
- `JxtaSocket socket` — the JxtaSocket used to connect to JxtaServerSocket

`main()`

This method creates a new SocketClient object to instantiate the JXTA platform and create the default net peer group, and then calls the run method to establish a connection to receive and send data.

```
    public static void main(String args[]) {

        try {
            String value = System.getProperty("RDVWAIT", "false");
            boolean waitForRendezvous =
                      Boolean.valueOf(value).booleanValue();
            SocketClient socEx = new SocketClient(waitForRendezvous);
            for (int i=0; i<3; i++) {
                socEx.run();
            }
            socEx.stop();
```

`startJxta()`

This method creates a configuration from a default configuration, and then instantiates the JXTA platform and creates the default net peer group :

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

Source Code: SocketClient

```
package tutorial.socket;

import net.jxta.peergroup.PeerGroup;
import net.jxta.platform.NetworkManager;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.socket.JxtaSocket;

import java.io.*;
import java.util.Arrays;
import java.text.MessageFormat;

/**
 * This tutorial illustrates the use JxtaSocket. It attempts to bind a
 * JxtaSocket to an instance of JxtaServerSocket bound socket.adv.
 * <p/>
 * Once a connection is established data is exchanged with the server.
 * The client will identify how many ITERATIONS of PAYLOADSIZE buffers will be
 * exchanged with the server and then write and read those buffers.
 */
public class SocketClient {
    /**
     * number of runs to make
     */
    private final static long RUNS = 8;

    /**
     * number of iterations to send the payload
     */
    private final static long ITERATIONS = 1000;

    /**
     * payload size
     */
    private final static int PAYLOADSIZE = 64 * 1024;

    private transient NetworkManager manager = null;

    private transient PeerGroup netPeerGroup = null;
    private transient PipeAdvertisement pipeAdv;
    private transient boolean waitForRendezvous = false;

    public SocketClient(boolean waitForRendezvous) {
        try {
            manager = new NetworkManager(NetworkManager.ConfigMode.ADHOC,
                    "SocketClient", new File(new File(".cache"),
"SocketClient").toURI());
            manager.startNetwork();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }

        netPeerGroup = manager.getNetPeerGroup();
        pipeAdv = SocketServer.createSocketAdvertisement();
        if (waitForRendezvous) {
            manager.waitForRendezvousConnection(0);
        }
    }

    /**
     * Interact with the server.
```

```
     */
    public void run() {
        try {

            if (waitForRendezvous) {
                manager.waitForRendezvousConnection(0);
            }

            long start = System.currentTimeMillis();
            System.out.println("Connecting to the server");
            JxtaSocket socket = new JxtaSocket(netPeerGroup,
                    //no specific peerid
                    null,
                    pipeAdv,
                    //connection timeout: 5 seconds
                    5000,
                    // reliable connection
                    true);

            // get the socket output stream
            OutputStream out = socket.getOutputStream();
            DataOutput dos = new DataOutputStream(out);

            // get the socket input stream
            InputStream in = socket.getInputStream();
            DataInput dis = new DataInputStream(in);

            long total = ITERATIONS * (long) PAYLOADSIZE * 2;
            System.out.println("Sending/Receiving " + total + " bytes.");

            dos.writeLong(ITERATIONS);
            dos.writeInt(PAYLOADSIZE);

            long current = 0;
            while (current < ITERATIONS) {
                byte[] out_buf = new byte[PAYLOADSIZE];
                byte[] in_buf = new byte[PAYLOADSIZE];

                Arrays.fill(out_buf, (byte) current);
                out.write(out_buf);
                out.flush();
                dis.readFully(in_buf);
                assert Arrays.equals(in_buf, out_buf);
                current++;
            }
            out.close();
            in.close();

            long finish = System.currentTimeMillis();
            long elapsed = finish - start;
            System.out.println(MessageFormat.format("EOT. Processed {0} bytes
in {1} ms.
                                    Throughput = {2} KB/sec.",
                                    total, elapsed, (total / elapsed) * 1000 /
1024));

            socket.close();
            System.out.println("Socket connection closed");
        } catch (IOException io) {
            io.printStackTrace();
        }
    }

    private void stop() {
```

```
            manager.stopNetwork();
        }

        /**
         * If the java property RDVWAIT set to true then this demo
         * will wait until a rendezvous connection is established before
         * initiating a connection
         *
         * @param args none recognized.
         */
        public static void main(String args[]) {
            /*
            System.setProperty("net.jxta.logging.Logging", "FINEST");
            System.setProperty("net.jxta.level", "FINEST");
            System.setProperty("java.util.logging.config.file",
"logging.properties");
            */
            try {
                Thread.currentThread().setName(SocketClient.class.getName() +
".main()");
                String value = System.getProperty("RDVWAIT", "false");
                boolean waitForRendezvous = Boolean.valueOf(value);
                SocketClient socEx = new SocketClient(waitForRendezvous);
                for (int i = 1; i <= RUNS; i++) {
                    System.out.println("Run #" + i);
                    socEx.run();
                }
                socEx.stop();
            } catch (Throwable e) {
                System.out.flush();
                System.err.println("Failed : " + e);
                e.printStackTrace(System.err);
                System.exit(-1);
            }
        }
    }
```

Chapter 16: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.

shows example output when the Server application is run, and shows example input from the Client application:

Example output: Server 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
```

Example output: Client 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 ....
```

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
- `\` — the pipe used to receive messages

`main()`

This method 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 :

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

Then it retrieves the discovery and pipe services :

```
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 :

```
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 :

```
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 :

```
discoSvc.publish(mcadv);
discoSvc.remotePublish(mcadv);
```

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 :

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

After the advertisement is created, we initialize the name, version, creator, ID, and URI :

```
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).

> 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.fromURI(new URI("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 :

```
FileInputStream is = new FileInputStream(FILENAME);
pipeadv = (PipeAdvertisement)
 AdvertisementFactory.newAdvertisement(
        MimeMediaType.XMLUTF8, is);
is.close();
```

After we successfully create our pipe advertisement, we add it to the module spec advertisement :

```
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 , and then we publish it to our local cache and propagate it to our rendezvous peer :

```
discoSvc.publish(mdadv);
discoSvc.remotePublish(mdadv);
```

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

```
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() :

```
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 :

```
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]:

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

and then continue to wait for the next message.

Source Code: Service Server

```
import java.io.StringWriter;
import java.io.FileInputStream;
import net.jxta.discovery.DiscoveryService;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.Element;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.document.StructuredDocumentFactory;
import net.jxta.document.StructuredDocumentUtils;
import net.jxta.document.StructuredTextDocument;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.MessageElement;
import net.jxta.exception.PeerGroupException;
import net.jxta.id.ID;
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.ModuleClassAdvertisement;
import net.jxta.protocol.ModuleSpecAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;
import net.jxta.protocol.PipeAdvertisement;


public class Server   {

    static PeerGroup group = null;
    static PeerGroupAdvertisement groupAdvertisement = null;
    private DiscoveryService discovery;
    private PipeService pipes;
    private InputPipe myPipe; // input pipe for the service
    private Message msg;      // pipe message received
    private ID gid;  // group id

    public static void main(String args[]) {
        Server myapp = new Server();
        System.out.println ("Starting Service Peer ....");
        myapp.startJxta();
        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);
        }

        // this is how to obtain the group advertisement
        groupAdvertisement = group.getPeerGroupAdvertisement();

        // get the discovery, and pipe service
        System.out.println("Getting DiscoveryService");
        discovery = group.getDiscoveryService();
        System.out.println("Getting PipeService");
```

```
        pipes = group.getPipeService();
        startServer();
    }

    private void startServer() {

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

        // get the peergroup service we need
        gid = group.getPeerGroupID();

        try {

            // First create the Service Module class advertisement
            // build the module class advertisement using the
            // AdvertisementFactory by passing the Advertisement type
            // we want to construct. The Module class advertisement
            // is to be used to simply advertise the existence of
            // the service. This is a very small advertisement
            // that only advertise the existence of service
            // In order to access the service, a peer must
            // discover the associated module spec advertisement.
            ModuleClassAdvertisement mcadv = (ModuleClassAdvertisement)
                        AdvertisementFactory.newAdvertisement(
                        ModuleClassAdvertisement.getAdvertisementTyp());

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

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

            // Once the Module Class advertisement is created, publish
            // it in the local cache and within the peergroup.
            discovery.publish(mcadv);
            discovery.remotePublish(mcadv);

            // Create the Service Module Spec advertisement
            // build the module Spec Advertisement using the
            // AdvertisementFactory class by passing in the
            // advertisement type we want to construct.
            // The Module Spec advertisement contains
            // all the information necessary for a client to reach
            // the service
            // i.e. it contains a pipe advertisement in order
            // to reach the service

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

            // Setup some information 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("JXTASPEC:JXTA-EX1");
            mdadv.setVersion("Version 1.0");
            mdadv.setCreator("sun.com");
            mdadv.setModuleSpecID(IDFactory.newModuleSpecID(mcID));
            mdadv.setSpecURI("http://www.jxta.org/Ex1");

            // Create the service pipe advertisement.
            // The client MUST use the same pipe advertisement to
```

```
        // communicate with the server. When the client
        // discovers the module advertisement it extracts
        // the pipe advertisement to create its pipe.
        // So, we are reading the advertisement from a default
        // config file to ensure that the
        // service will always advertise the same pipe
        //
        System.out.println("Reading in pipeserver.adv");
        PipeAdvertisement pipeadv = null;

        try {
            FileInputStream is = new FileInputStream(
                                        "pipeserver.adv");
            pipeadv = (PipeAdvertisement)
                      AdvertisementFactory.newAdvertisement(
                          MimeMediaType.XMLUTF8, is);
            is.close();
        } catch (Exception e) {
            System.out.println("failed to read/parse pipe
                                          advertisement");
            e.printStackTrace();
            System.exit(-1);
        }

        // Store the pipe advertisement in the spec adv.
        // This information will be retrieved by the client when it
        // connects to the service
        mdadv.setPipeAdvertisement(pipeadv);

        // display the advertisement as a plain text document.
        StructuredTextDocument doc = (StructuredTextDocument)
                                     mdadv.getDocument
                                          (MimeMediaType.XMLUTF8);

        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.
        discovery.publish(mdadv);
        discovery.remotePublish(mdadv);

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

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

    // Ok no way to stop this daemon, but that's beyond the point
    // of the example!
    while (true) { // loop over every input received from clients

        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;
                }

                // Read the message as a String
                String ip = null;

                try {

                    // NOTE: The Client and Service must agree on the tag
                    // names. This is part of the Service protocol defined
                    // to access the service.
                    // get all the message elements
                    Message.ElementIterator en = msg.getMessageElements();
                    if (!en.hasNext()) {
                        return;
                    }
                    // get the message element named SenderMessage
                    MessageElement msgElement = msg.getMessageElement(null, "DataTag");
                    // Get message
                    if (msgElement.toString() != null) {
                        ip = msgElement.toString();
                    }

                    if (ip != null) {
                       // read the data
                       System.out.println("Server: receive message: " + ip);

                    } else {
                      System.out.println("Server: error could not find the tag");
                    }
                } catch (Exception e) {
                    System.out.println("Server: error receiving message");
                }
            }
        }
    }
```

Example Service Advertisement: `pipeserv er.ad v` 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>
```

Service 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 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 :

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

Then it retrieves the discovery and pipe services :

```
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 :

```
en = 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 :

```
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 :

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

We print the advertisement as a plain text document and then extract the pipe advertisement from the module spec advertisement :

```
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() :

```
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 :

```
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 :

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

Source Code: Service Client

```
import java.io.IOException;
import java.io.StringWriter;
import java.net.MalformedURLException;
import java.net.URL;
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.document.TextElement;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.StringMessageElement;
import net.jxta.exception.PeerGroupException;
import net.jxta.id.IDFactory;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.pipe.OutputPipe;
import net.jxta.pipe.PipeID;
import net.jxta.pipe.PipeService;
import net.jxta.protocol.ModuleSpecAdvertisement;
import net.jxta.protocol.PeerGroupAdvertisement;
import net.jxta.protocol.PipeAdvertisement;


/**
 *  Client Side: This is the client side of the JXTA-EX1
 *  application. The client application is a simple example on how to
 *  start a client, connect to a JXTA enabled service, and invoke the
 *  service via a pipe advertised by the service. The
 *  client searches for the module specification advertisement
 *  associated with the service, extracts the pipe information to
 *  connect to the service, creates a new output to connect to the
 *  service and sends a message to the service.
 *  The client just sends a string to the service no response
 *  is expected from the service.
 */

public class Client {

    static PeerGroup netPeerGroup = null;
    static PeerGroupAdvertisement groupAdvertisement = null;
    private DiscoveryService discovery;
    private PipeService pipes;
    private OutputPipe myPipe; // Output pipe to connect the service
    private Message msg;

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

        // this is how to obtain the group advertisement
        groupAdvertisement = netPeerGroup.getPeerGroupAdvertisement();
        // get the discovery, and pipe service
        System.out.println("Getting DiscoveryService");
        discovery = netPeerGroup.getDiscoveryService();
        System.out.println("Getting PipeService");
        pipes = netPeerGroup.getPipeService();
        startClient();
    }

    // start the client
    private void startClient() {

        // Let's initialize the client
        System.out.println("Start the Client");

        // Let's try to locate the service advertisement
        // we will loop until we find it!
        System.out.println("searching for the JXTA-EX1 Service
                                            advertisement");
        Enumeration en = null;
        while (true) {
            try {

                // let's look first in our local cache to see
                // if we have it! We try to discover an adverisement
                // which as the (Name, JXTA-EX1) tag value

                en = discovery.getLocalAdvertisements
                                    (DiscoveryService.ADV,
                                     "Name",
                                     "JXTASPEC:JXTA-EX1");

                // Ok we got something in our local cache does not
                // need to go further!
                if ((en != null) && en.hasMoreElements()) {
                    break;
                }

                // nothing in the local cache?, let's remotely query
                // for the service advertisement.
                discovery.getRemoteAdvertisements(null,
                                                  DiscoveryService.ADV,
                                                  "Name",
                                                  "JXTASPEC:JXTA-EX1",
                                                  1, null);

                // The discovery is asynchronous as we do not know
                // how long is going to take
                try { // sleep as much as we want. Yes we
                    // should implement asynchronous listener pipe...
                    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 Spec Advertisement
            ModuleSpecAdvertisement mdsadv = (ModuleSpecAdvertisement)
                                                    en.nextElement();
            try {
                // let's print the advertisement as a plain text document
                StructuredTextDocument doc = (StructuredTextDocument)
                                        mdsadv.getDocument
                            (MimeMediaType.TEXT_DEFAULTENCODING);

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

                // we can find the pipe to connect to the service
                // in the advertisement.
                PipeAdvertisement pipeadv = mdsadv.getPipeAdvertisement();

                // Ok we have our pipe advertiseemnt to talk to the service
                // 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
                for (int i=0; i<3; i++) {
                    myPipe = pipes.createOutputPipe(pipeadv, 10000);
                }

                // create the data string to send to the server
                String data = "Hello my friend!";

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

Chapter 17:Password Protected Peer Group

This example[11] illustrates how to create and join a new peer group that implements authentication via a login and a password.

shows example output when this application is run:

Example output: Creating and joining a peer group that requires authentication.

```
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>
                PrivatePeerGroups:FZUH:
             </login>
          </Parm>
        </Svc>
</jxta:PGA>
```

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: PrivatePeerGroups
- Password: RULE

11 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.

`main()`

This method call the constructor PrivatePeerGroup() of the class and instantiates a new PrivatePeerGroup Object called satellaRoot.

The constructor method `PrivatePeerGroup()`

This method creates and joins the private 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
- Instantiates the user login, password, group name and group ID
- Creates the authenticated peer group called "SatellaGroup" by calling the createPeerGroup() method
- Searches for the "SatellaGroup" peer group by calling the discoverPeerGroup() method
- Joins the "SatellaGroup" peer group by calling the joinPeerGroup() method
- Prints on standard output the XML Advertisement of the "SatellaGroup" peer group by calling the printXmlAdvertisement() method

`startJxta()`

This method instantiates the JXTA platform, creates (and later returns) the default netPeerGroup called myNetPeeGroup :

```
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 :

```
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 :

```
rootPeerGroupDiscoveryService.publish(passwdMembershipModuleImplAdv
                                      PeerGroup.DEFAULT_LIFETIME,

                                      PeerGroup.DEFAULT_EXPIRATION);
rootPeerGroupDiscoveryService.remotePublish(passwdMembershipModuleImplAdv,
                                            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,:

```
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 :

```
rootPeerGroupDiscoveryService.publish(satellaPeerGroupAdv,
                                      PeerGroup.DEFAULT_LIFETIME,
                                      PeerGroup.DEFAULT_EXPIRATION);
rootPeerGroupDiscoveryService.remotePublish(satellaPeerGroupAdv,
                                            PeerGroup.DEFAULT_EXPIRATION);
```

Finally the peer group is created from the peer group advertisement :

```
satellaPeerGroup=rootPeerGroup.newGroup(satellaPeerGroupAdv);
```

And returned :

```
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). 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 :
```
allPurposePeerGroupImplAdv=
    rootPeerGroup.getAllPurposePeerGroupImplAdvertisement();
```
- Extract its "Param" element. As mentioned above, this contains the services provided by the peer group :
```
passwdMembershipPeerGroupParamAdv =
    new StdPeerGroupParamAdv(allPurposePeerGroupImplAdv.getParam());
```

From this "Param" element, extract the peer group services and their associated service IDs :
```
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 :
```
if (allPurposePeerGroupServiceID.equals(PeerGroup.membershipClassID))
```
- Once found, extract the generic membership service advertisement :
```
ModuleImplAdvertisement allPurposePeerGroupMemershipServiceModuleImplAdv=
    (ModuleImplAdvertisement)
    allPurposePeerGroupServicesHashtable.get(allPurposePeerGroupServiceID);
```
- Use this generic advertisement to generate a custom one for the Password Membership Service using the createPasswdMembershipServiceModuleImplAdv() method :
```
passwdMembershipServiceModuleImplAdv=
    this.createPasswdMembershipServiceModuleImplAdv
    (allPurposePeerGroupMemershipServiceModuleImplAdv);
```
- Remove the generic Membership advertisement :
```
allPurposePeerGroupServicesHashtable.remove(allPurposePeerGroupServiceID);
```
- And replace it by the new one :
```
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 :
```
passwdMembershipPeerGroupModuleImplAdv.setParam(
    (Element)PasswdMembershipPeerGroupParamAdv.getDocument(new
       MimeMediaType("text/xml")));
```
- And Update the Password Membership peer group module implementation advertisement spec ID . 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.fromURI(new URI("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.

lists the generic advertisement that is receives as argument by this method.

0XML representation of a typical MembershipService, extracted from the Parm element of a peer group module implementation advertisement.

```
<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
      </Desc>
  </jxta:MIA>
</Svc>
```

This method needs only to update the Module Spec ID, the code, and description with values specific to the PasswdMembershipService :

```
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 :

```
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 :

```
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 :

```
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
```
satellaPeerGroupAdv.setModuleSpecID(
passwdMembershipModuleImplAdv.getModuleSpecID());
```
- Its name and description :
```
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 shows the creation of this Service Parameters XML document:
```
StructuredTextDocument loginAndPasswd= (StructuredTextDocument)
StructuredDocumentFactory.newStructuredDocument(new MimeMediaType
   ("text/xml"),"Parm");
```

Whereas lines - 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) :
```
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 :
```
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 :
```
satellaPeerGroup=myNetPeerGroup.newGroup(satellaPeerGroupAdv);
```

`joinPeerGroup()`

This method is very similar to the joinGroup() method described earlier (see “ ” on page 111). 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 :
```
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 :
```
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")

And goes through all the Authentication method place into looking for "setAuth1Identity" and "setAuth2_Password" and invokes them with the appropriate parameters:
```
   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: PrivatePeerGroup

```
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;

import net.jxta.impl.peergroup.StdPeerGroupParamAdv ;

public class PrivatePeerGroup  {

    private PeerGroup  myNetPeerGroup=null,
satellaPeerGroup=null,discoveredSatellaPeerGroup=null;
    private final static PeerGroupID satellaPeerGroupID = PeerGroupID.create(;
         URI.create("jxta:uuid-4d6172676572696e204272756e6f202002"));


    /** Creates new RootWS */
    public PrivatePeerGroup() {
        // Starts the JXTA Platform
        myNetPeerGroup=this.startJxta();
        if (myNetPeerGroup!=null) {
            System.out.println("JXTA platform Started ...");
        } else {
            System.err.println("Failed to start JXTA : 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="PrivatePeerGroups";
        String passwd="RULE";
        String groupName="SatellaGroup";


        // 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 satellaPeerGroupAdvertisement;

        // Create the PeerGroup Module Implementation Adv
        ModuleImplAdvertisement passwdMembershipModuleImplAdv;
        passwdMembershipModuleImplAdv=this.createPasswdMembershipPeerGroupModuleImplAdv(ro
otPeerGroup);
        // Publish it in the parent peer group
        DiscoveryService rootPeerGroupDiscoveryService =
                                            rootPeerGroup.getDiscoveryService();
        try {
            rootPeerGroupDiscoveryService.publish(
                      passwdMembershipModuleImplAdv,
                      PeerGroup.DEFAULT_LIFETIME,
                      PeerGroup.DEFAULT_EXPIRATION);
            rootPeerGroupDiscoveryService.remotePublish(
                           passwdMembershipModuleImplAdv,
                              PeerGroup.DEFAULT_EXPIRATION);
        } catch (java.io.IOException e) {
            System.err.println("Can't Publish passwdMembershipModuleImplAdv");
            System.exit(1);
        }
        // Now, Create the Peer Group Advertisement
        satellaPeerGroupAdvertisement=
            this.createPeerGroupAdvertisement(passwdMembershipModuleImplAdv,
                                                        groupName,login,passwd);
        // Publish it in the parent peer group
        try {
            rootPeerGroupDiscoveryService.publish(
                      satellaPeerGroupAdvertisement,
                      PeerGroup.DEFAULT_LIFETIME,
                      PeerGroup.DEFAULT_EXPIRATION);
            rootPeerGroupDiscoveryService.remotePublish(
                                satellaPeerGroupAdvertisement,
                                   PeerGroup.DEFAULT_EXPIRATION);
        } catch (java.io.IOException e) {
            System.err.println("Can't Publish  satellaPeerGroupAdvertisement");
            System.exit(1);
        }
```

```
        // Finally Create the Peer Group
        if (satellaPeerGroupAdvertisement==null) {
            System.err.println("satellaPeerGroupAdvertisement is null");
        }
        try {
            satellaPeerGroup=rootPeerGroup.newGroup(satellaPeerGroupAdvertisement);
        } 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  satellaPeerGroupAdvertisement=
                  (PeerGroupAdvertisement)
                  AdvertisementFactory.newAdvertisement(
                        PeerGroupAdvertisement.getAdvertisementType());

        // Instead of creating a new group ID each time, by using the
        //  line below
        // satellaPeerGroupAdvertisement.setPeerGroupID
        //               (IDFactory.newPeerGroupID());
        // I use a fixed ID so that each time I start PrivatePeerGroup,
        // it creates the same Group
        satellaPeerGroupAdvertisement.setPeerGroupID(
                                  satellaPeerGroupID);
        satellaPeerGroupAdvertisement.setModuleSpecID(
                    passwdMembershipModuleImplAdv.getModuleSpecID());
        satellaPeerGroupAdvertisement.setName(groupName);
        satellaPeerGroupAdvertisement.setDescription("Peer Group using Password
Authentication");

        // Now create the Structured Document Containing the login and
        // passwd informations. 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
            satellaPeerGroupAdvertisement.putServiceParam(
                         PeerGroup.membershipClassID,loginAndPasswd);
        }
        return satellaPeerGroupAdvertisement;
    }

    private ModuleImplAdvertisement createPasswdMembershipPeerGroupModuleImplAdv(PeerGroup
rootPeerGroup) {
```

```
        // Create a ModuleImpl Advertisement for the Passwd
        // Membership Service  Take a allPurposePeerGroupImplAdv
        //  ModuleImplAdvertisement parameter to
        // Clone some of its fields. It is easier than to recreate
        // 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.getAllPurposePeerGroupImplAdvertiseme
nt();
        } catch (java.lang.Exception e) {
            System.err.println("Can't Execute:
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.createPasswdMembershipServiceMod
uleImplAdv(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(MimeMediaType.XMLUTF));
                // Update its Spec ID This comes from the
                // Instant P2P PeerGroupManager Code (Thanks !!!!)
                if (!passwdMembershipPeerGroupModuleImplAdv.getModuleSpecID().equals(
                                    PeerGroup.allPurposePeerGroupSpecID)) {
                    passwdMembershipPeerGroupModuleImplAdv.setModuleSpecID(IDFactory.newMo
duleSpecID(
                                passwdMembershipPeerGroupModuleImplAdv.getModuleSpecID().g
etBaseClass()));
                } else {
                    ID passwdGrpModSpecID= ID.create( URI.create(
                         "urn" + "jxta:uuid-"+
                      "DeadBeefDeafBabaFeedBabe00000001" +"04" +"06"));

                    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.passw
ordMembershipSpecID);
        passwdMembershipServiceModuleImplAdv.setCode(PasswdMembershipService.class.getName
());
        passwdMembershipServiceModuleImplAdv.setDescription(" Module Impl Advertisement
for the PasswdMembership Service");
        passwdMembershipServiceModuleImplAdv.setCompat(allPurposePeerGroupMemershipServiceModuleImp
dv.getCompat());
        passwdMembershipServiceModuleImplAdv.setUri(allPurposePeerGroupMemershipServiceModuleI
mplAdv.getUri());
        passwdMembershipServiceModuleImplAdv.setProvider(allPurposePeerGroupMemershipServiceModuleI
lAdv.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 asynchroneously.
        // 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 it's parent is null");
            System.exit(1);
        }
        boolean isGroupFound=false;
        Enumeration localPeerGroupAdvertisementEnumeration = null;
        PeerGroupAdvertisement satellaPeerGroupAdvertisement = 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)) {
                        satellaPeerGroupAdvertisement=pgAdv;
                        isGroupFound=true ;
                        break ;
                    }
                }
            }
            try {
                Thread.sleep(5 * 1000);
            } catch(Exception e) {}
        }
        try {
            satellaPeerGroup=myNetPeerGroup.newGroup(satellaPeerGroupAdvertisement);
        } 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 with fields of the application needs 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
        System.out.println(adv.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 instantiate 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[]) {
        PrivatePeerGroup satellaRoot = new PrivatePeerGroup();
        System.exit(0);
    }
}
```

GLOSSARY

Advertisement

JXTA's language-neutral meta-data 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).

Codat

The combination of a content (commonly a document or file) and a JXTA ID.

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 .

J2SE

Java 2 Platform, Standard Edition software. See Java SE Overview

JXTA

JXTA is not an acronym, and in particular the "J"does not refer to Java. JXTA is a made up word coined by the project's original sponsor, Bill Joy. JXTA is derived from the word Juxtapose, as in side by side. It is a recognition that peer-to-peer is juxtaposed to client server or Web based computing -- what is considered today's traditional computing model.

jux ta pose

*tr. v.*To place side by side, especially for comparison or contrast.

Message

The basic unit of data exchange between JXTA 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.

Network Peer Group

The base peer group for applications and services within the JXTA network. Most applications and services will instantiate their own peer groups using the Network Peer Group as a base.

Output Pipe

A pipe endpoint; the sending end of a pipe. Pipe endpoints are dynamically bound to peer endpoints at runtime.

Peer-to-Peer (P2P)

A decentralized networking paradigm in which distributed nodes, or peers, communicate and work collaboratively to provide services.

PBP

Pipe 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 node

Maintains information on routes to other peers, and helps relay messages to peers.

Rendezvous node

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.)

World Peer Group, PlatformGroup

The most fundamental peer group within the JXTA network. This core peer group is generally responsible only for management of physical network connections, physical network (generally broadcast) discovery and physical network topology management. Applications generally do not interact with this group. May only include limited endpoint, resolver, discovery and rendezvous services.

Appendix I: References

The following Web pages contain information on Project JXTA:

- http://www.jxta.org — home Web page for Project JXTA
- http://jxta-spec.dev.java.net/ — Project JXTA specification
- *http://jxta-jxse.dev.java.net* — Project JXTA platform infrastructure and protocols for the J2SE platform binding
- http://platform.jxta.org/java/api/overview-tree.html — public API (Javadoc)
- 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.
- PKI Security for JXTA Overlay Networks, **Jeffrey Eric Altman, IAM Consulting.**

Appendix II: 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.

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 , 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://jxse-shell.dev.java.net

Starting from a clean state

n 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.
The JavaDoc for the `net.jxta.logging.Logging` class provides complete instructions and examples for enabling detailed logging information.

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 `cm` directory (located in the under $JXTA_HOME directory, by default .jxta) and then re-run the application.