JXTA v2.3.x: Java™ Programmer's Guide

November 21, 2005

Table of Contents

Chapter 1: Introduction....7
Why JXTATM ?....7
What can be done with JXTATM Technology? 8
Where to get the JXTATM technology....9
Getting Involved....9
Chapter 2: JXTATM Architecture....10
Overview....10
JXTA Components....11
Key aspects of the JXTA architecture....11
Chapter 3: JXTA Concepts....12
Peers....12
Peer Groups....12
Network Services 14
Modules....14
Pipes....15
Bidirectional reliable communication channels (JxtaSocket, and JxtaBiDiPipe)....17
Messages....18
Advertisements....19
Security....20
Security....20
IDs....21
Chapter 4: Network Architecture....22
Network Organization....22
Shared Resource Distributed Index (SRDI)....23
Queries....24
Firewalls and NAT....25
Chapter 5: JXTA Protocols....27
Peer Discovery Protocol....28
Peer Information Protocol....28
Peer Resolver Protocol....28
Pipe Binding Protocol....29
Endpoint Routing Protocol....29
Rendezvous Protocol....30
Chapter 6: Hello World Example....31
Getting Started....31
Accessing On-line Documentation 31
Downloading Binaries....31
Compiling JXTA Code....32
Configuration....33
HelloWorld Example....35
Running the Hello World Example....36
Source Code: SimpleJxtaApp....38
Chapter 7: Programming with JXTA....38

Peer Discovery....39
Discovery Service....39
DiscoveryDemo....40
Source Code: DiscoveryDemo....43
Peer Group Discovery....45
Source Code: GroupDiscoveryDemo....47
Creating Peer Groups and Publishing Advertisements....50
groupsInLocalCache()....50
createGroup()....51
Source Code: PublishDemo....52
Joining a Peer Group....54
Membership Service....54
createGroup()....55
joinGroup()....55
Source Code: JoinDemo....57
Sending Messages Between Two Peers....60
JXTA Pipe Service....60
PipeListener....61
pipeMsgEvent()....62
Source Code: PipeListener....63
PipeExample....67
outputPipeEvent() 67
rendezvousEvent()....68
Source Code: PipeExample....69
examplepipe.adv....72
Using a JxtaBiDiPipe (A bidirectional reliable pipe)....73
JxtaBiDiPipe....73
JxtaServerPipeExample....74
Source Code: JxtaServerPipeExample....75
Example pipe advertisement: pipe.adv....79
JxtaBidiPipeExample....80
Source Code: JxtaBidiPipeExample....81
Using JxtaSockets (bidirectional reliable pipes with java.net.Socket interface)....86
JxtaServerSocketExample....87
Source Code: JxtaServerSocketExample....88
Example pipe advertisement: socket.adv....91
JxtaSocketExample....92
Source Code: JxtaSocketExample....93
JXTA Services....98
Creating a JXTA Service....99
Server....101
readMessages()....103
Source Code: Server....104
Example Service Advertisement:....109
Client....110

Source Code: Client....112
The constructor method SecurePeerGroup()....117
createPeerGroup()....117
createPasswdMembershipPeerGroupModuleImplAdv ()....118
createPeerGroupAdvertisement()....120
discoverPeerGroup()....120
joinPeerGroup()....120
completeAuth()....121
Source Code: SecurePeerGroup....122
Chapter 8: JXTA Extension Package....133
Chapter 9: 146
Chapter 9: 146
....146
Miscellaneous....146
....146
....146
TestingConfigurator.java: 154
Authenticating via membership APIs 169
Creating the advertisement XML 173
Creating the FooAdv Advertisement base class 173
Creating the BarAdv Advertisement subclass 176
How Advertisement instantiation works 178
How do you know that JXTA is working? 180
Checking with your Browser 182
Use telnet to test the TCP port 182
Warnings and Errors 182
JXTA Network Stability 182
What is a Well Known ID? 192
What are Well Known IDs normally used for? 192
Are there alternatives to Well Known IDs? 192
Are there problems with Well Known IDs? 192
Avoiding ID collisions 192
Playing it Safe 193
A few links for examples and other information 193
Calculating a Well Known ID 193
Chapter 10: References....199
Glossary....200
Troubleshooting....204
Errors compiling JXTA applications....204
Errors running JXTA applications....204
Unable to discover JXTA peers....204
Using the JXTA Shell....204
Starting from a clean state....205
Displaying additional log information....205
Removing User name or Password....206

Chapter 1: Introduction

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

This document is intended for software developers who would like to write and deploy P2P services and applications using the Java programming language and JXTA technology. It provides an introduction to the JXTA technology, describes the JXTA network architecture and key concepts, and includes examples and discussion of essential programming constructs using the JXTA platform J2SE 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 popular. Popular software based on P2P technologies includes file sharing, distributed computing, and instant messenger services. While each of these applications performs different tasks, they all share many of the same properties, such as discovery of peers, searching, and file or data transfer. Currently, application development is inefficient, with developers solving the same problems and duplicating similar infrastructure implementation. And, most applications are specific to a single platform and are unable to communicate and share data with other applications.

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

- One primary goal of JXTA is to provide a platform with the basic functions necessary for a P2P network. In addition, JXTA technology seeks to overcome potential shortcomings in many of the existing P2P systems:
- *Interoperability* — JXTA technology is designed to enable peers providing various P2P services to locate each other and communicate with each other.
- *Platform independence* — JXTA technology is designed to be independent of programming languages, transport protocols, and deployment platforms.

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

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

What is JXTA™?

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

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

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

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

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

What can be done with JXTA™ Technology?

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

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

Where to get the JXTA™ technology

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

Getting Involved

As with any open source project, a primary goal is to get the community involved by contributing to 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 chat with other JXTA users and contributors using the myJXTA2 application which can be downloaded at:

  http://download.jxta.org/easyinstall/install.html.

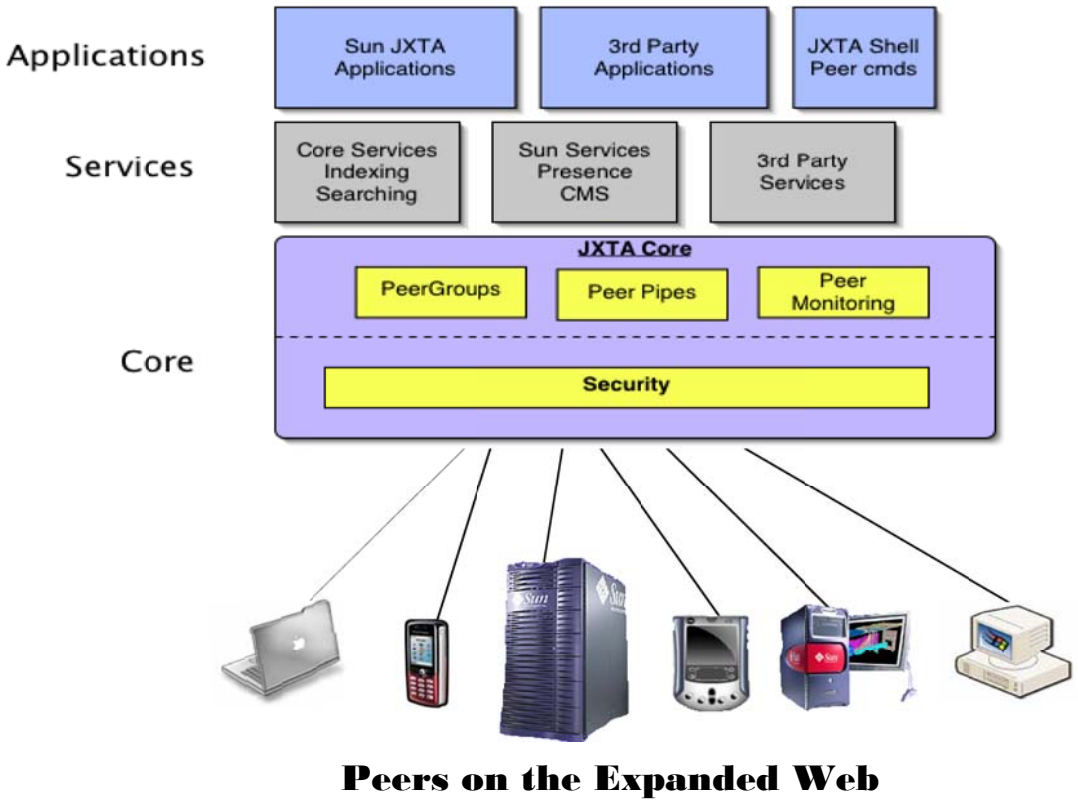
  The demonstration application is available for the following platforms: Microsoft Windows, Solaris™ Operating Environment, Linux, UNIX, Mac OS X, and other Java technology enabled platforms

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

Chapter 2: JXTA™ Architecture

Overview

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



Platform Layer (JXTA Core)

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

- *Services Layer*

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

- *Applications Layer*

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

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

JXTA Components

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

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

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

These concepts are described in detail in the following chapters.

Key aspects of the JXTA architecture

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

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

Chapter 3: JXTA Concepts

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

Peers

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

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

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

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

Peer Groups

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

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

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

There are several motivations for creating peer groups:

- *To create a secure environment*

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

- *To create a scoping environment*

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

- *To create a monitoring environment*

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

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

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

The core peer group services include the following:

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

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

Network Services

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

The JXTA protocols recognize two levels of network services:

- *Peer Services*

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

- *Peer Group Services*

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

Services can be either pre-installed onto a peer or loaded from the network. In order to actually run a service, a peer may have to locate an implementation suitable for the peer's runtime environment. The process of finding, downloading, and installing a service from the network is similar to performing a search on the Internet for a Web page, retrieving the page, and then installing the required plug-in.

Modules

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

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

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

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

- *Module Class*

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

- *Module Specification*

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

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

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

- *Module Implementation*

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

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

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

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

Pipes

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

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

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

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

- *Point-to-point Pipes*

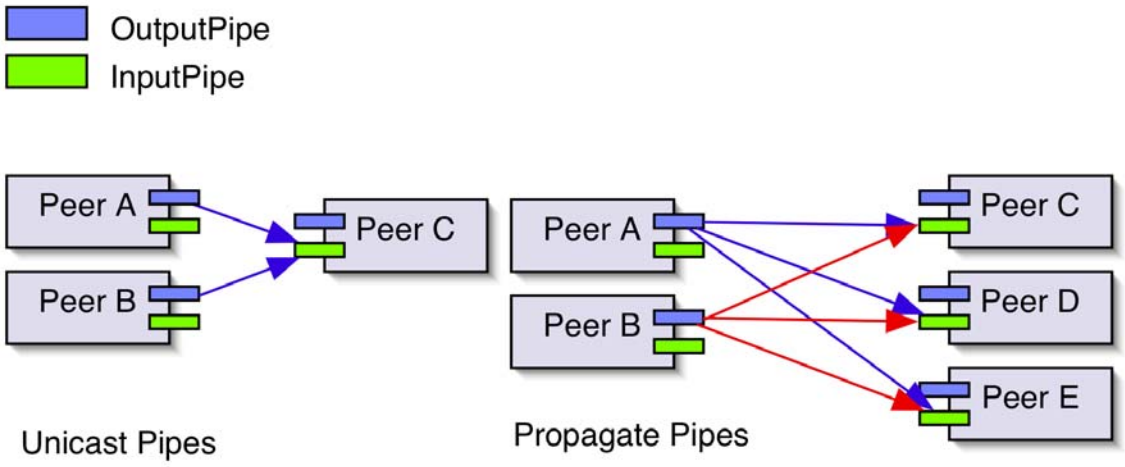
  A point-to-point pipe connects exactly two pipe endpoints together: an input pipe on one peer receives messages sent from the output pipe of another peer, it is also possible for multiple peers to bind to a single input pipe.

- *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. All propagation is done within the scope of a peer group. That is, the output pipe and all input pipes must belong to the same peer group.

- *Secure Unicast Pipes*

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

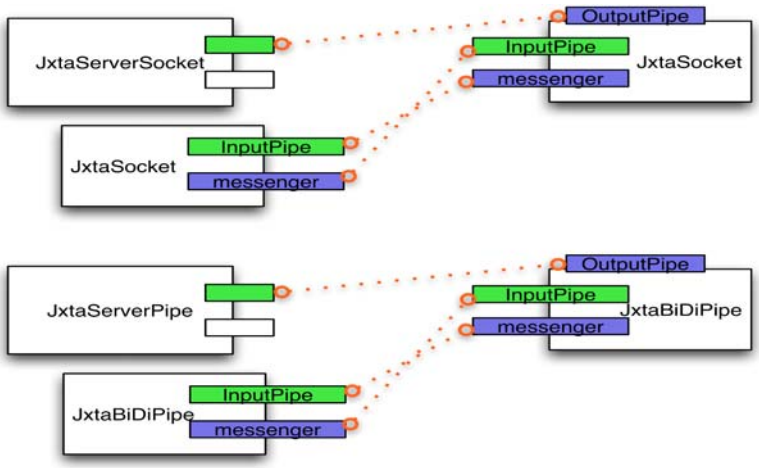


Bidirectional reliable communication channels (JxtaSocket, and JxtaBiDiPipe)

Since pipes provide unidirectional, unreliable communication channels, it is necessary to implement bidirectional and reliable communication channels. The platform provides the following to address the level of service quality required by applications :

- Reliability Library
  - Ensures message sequencing
  - Ensures delivery
  - Exposes message, and stream interfaces
- JxtaSocket, JxtaServerSocket provides :
  - Sub-class java.net.Socket, and java.net.ServerSocket respectively
  - Built on top of pipes, endpoint messengers, and the reliability library
  - Provides bidirectional and reliable communication channels
  - Exposes stream based interface ala Socket
  - Provides configurable internal buffering, and message chunking
  - Does not implement the Nagels algorithm, therefore streams must be flushed as needed
- JxtaBiDiPipe, and JxtaServerPipe provides :
  - Built on top of pipes, endpoint messengers, and the reliability library
  - Provides bidirectional and reliable communication channels
  - Exposes message based interface
  - Provides no message chunking (applications need to ensure message size does not exceed the platform message size limitation of 64K)

JxtaServerSocket, and JxtaServerPipe expose a input pipe to process connection requests, and negotiate communication parameters, whereby JxtaSocket, and JxtaBiDpipe bind to respectively to establish private dedicated pipes independent of the connection request pipe.



Messages

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

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

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

There are two representations for messages: XML and binary. The JXTA J2SE platform binding uses a binary format envelop to encapsulate the message payload. Services can use the most appropriate format for that transport (e.g., a service which requires a compact representation for a messages can use the binary representation, while

other services can use XML). Binary data may be encoded using a Base64 encoding scheme in the body of an XML message.

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

Advertisements

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

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

The JXTA protocols define the following advertisement types:

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

- *Rendezvous Advertisement* — describes a peer that acts as a rendezvous peer for a given peer group.
- *Peer Info Advertisement* — describes the peer info resource. The primary use of this advertisement is to hold specific information about the current state of a peer, such as uptime, inbound and outbound message count, time last message received, and time last message sent.

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

An example of a pipe advertisement is included in .

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

<!DOCTYPE jxta:PipeAdvertisement>

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

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

Security

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

Five basic security requirements must be provided:

Figure 0*Confidentiality* — guarantees that the contents of a message are not disclosed to unauthorized individuals.

Figure 1*Authentication* — guarantees that the sender is who he or she claims to be.

Figure 2*Authorization* — guarantees that the sender is authorized to send a message.

Figure 3*Data integrity* — guarantees that the message was not modified accidentally or deliberately in transit.

Figure 4*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 guarantee the data integrity of messages. Messages may also be encrypted (using public keys) and signed (using certificates) for confidentiality and refutability. Credentials can be used to provide message authentication and authorization.

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

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

IDs

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

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

An example JXTA peer ID is:

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

An example JXTA pipe ID is:

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

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

1 See IETF RFC 2141 for more information on URNs.

Chapter 4: Network Architecture

Network Organization

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

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

- *Minimal edge peer*

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

- *Full-featured edge peer*

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

- *Rendezvous peer*

  A rendezvous peer is like any other peer, and maintains a cache of advertisements. However, rendezvous peers also forward discovery requests to help other peers discover resources. When a peer joins a peer group, it automatically seeks a rendezvous peer.[2] If no rendezvous peer is found, it dynamically becomes a rendezvous peer for that peer group. Each rendezvous peer maintains a list of other known rendezvous peers and also the peers that are using it as a rendezvous.

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

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

- *Relay peer*[3]

  A relay peer maintains information about the routes to other peers and routes messages to peers. A peer first looks in its local cache for route information. If it isn't found, the peer sends queries to relay peers asking for route information. Relay peers also forward messages on the behalf of peers that cannot

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

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

directly address another peer (e.g., NAT environments), bridging different physical and/or logical networks

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

Shared Resource Distributed Index (SRDI)

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

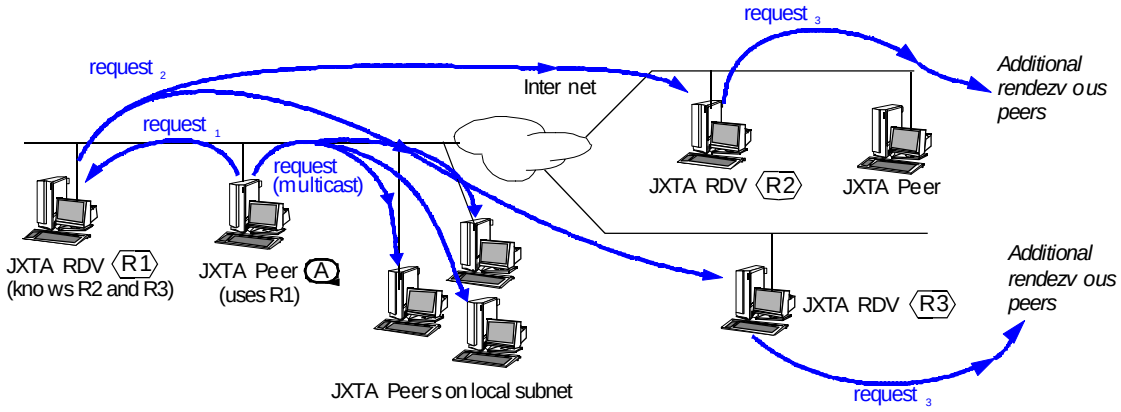
Each rendezvous maintains its own list of known rendezvous in the peer group. A rendezvous may retrieve rendezvous information from a pre-defined set of bootstrapping, or seeding, rendezvous. Rendezvous periodically select a given random number of rendezvous peers and send them a random list of their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

When a peer publishes a new advertisement, the advertisement is indexed by the SRDI service using keys such as the advertisement name or ID. Only the indices of the advertisement are pushed to the rendezvous by SRDI, minimizing the amount of data that needs to be stored on the rendezvous. The rendezvous also pushes the index to additional rendezvous peers (selected by the calculation of a hash function of the advertisement index). [4]

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.

Queries

An example configuration is shown in . 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



.

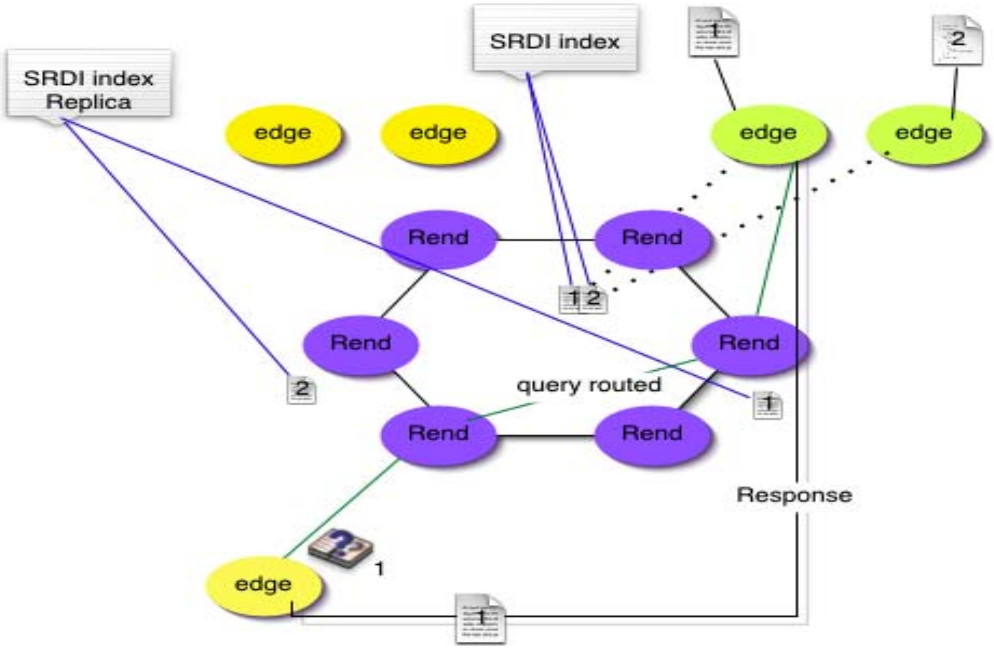
Request propagation via rendezvous peers.

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

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

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

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



Firewalls and NAT

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

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

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

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

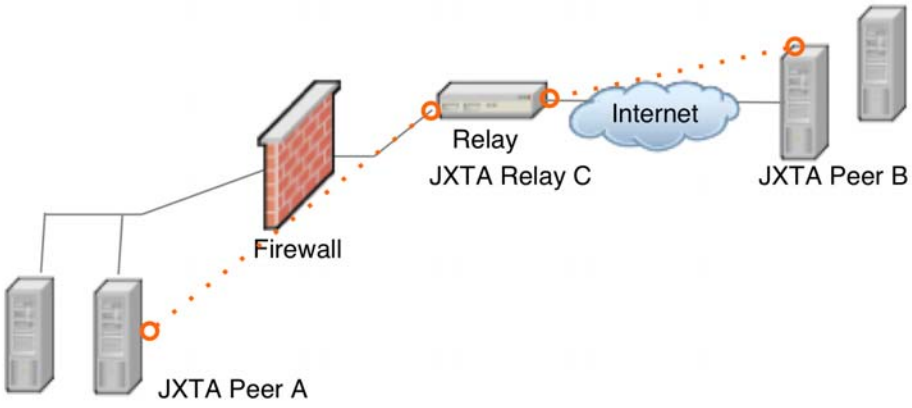


Figure 4-3 Message routing scenario across a firewall.

Chapter 5: JXTA Protocols

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

There are six JXTA protocols:

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

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

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

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

Peer Discovery Protocol

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

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

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

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

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

Peer Information Protocol

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

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

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

Peer Resolver Protocol

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

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

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

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

Pipe Binding Protocol

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

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

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

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

Endpoint Routing Protocol

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

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

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

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

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

Rendezvous Protocol

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

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

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

Chapter 6: Hello World Example

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

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

Getting Started

System Requirements

The current Project JXTA J2SE platform binding requires a platform that supports the Java Run-Time Environment (JRE) or Software Development Kit (SDK) 1.4.1 release or later. This environment is currently available on the Solaris Operating Environment, Microsoft Windows 95/98/2000/ME/NT 4.0, Linux, and Mac OS X.

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

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

Accessing On-line Documentation

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

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

Downloading Binaries

Download the Companion Tutorial 2.x Programs [6] at http://www.jxta.org/ProgGuideExamples.zip. The compressed archive contains all the JXTA platform and supporting libraries, sources and binaries of the tutorials covered in this guide, and run scripts.

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

Table 0*Release Builds* — the most recently saved stable build of the software; these are the best choice for new JXTA users. Easy to use installers for these builds are available by following the link on the Web page to the Project JXTA Easy Installers (http://download.jxta.org/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.

Table 1*Nightly Builds* — the automated builds of the current "work in progress"; these builds are provided for developer testing, and are not guaranteed to function correctly.

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

6 All of the covered tutorials are contained within the platform sources under platform/www/java/tutorial/examples along with build tools.

Compiling JXTA Code

The application in this example, SimpleJxtaApp, 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 this `.jar` file. For example, users on the Window systems could use a command similar to , substituting the actual location of the `jxta.jar` file on their system:

Example compilation command (Windows systems).

```
C:> javac -classpath .\lib\jxta.jar SimpleJxtaApp.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 (see). For example, users on Window systems could use a command similar to , substituting the actual location of their `.jar` files:

Example command to run application (Windows systems).

```
C:> java -classpath .\lib\jxta.jar;.\lib\log4j.jar;
     .\lib\bcprov-jdk14.jar;. SimpleJxtaApp
```

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

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. Alternatively, you may choose to set the `CLASSPATH` environment variable, which has the same effect as specifying the `-classpath` compilation option.

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. Alternatively, you may choose to set the `CLASSPATH` environment variable, which has the same effect as specifying the `-classpath` command line option.

Configuration

The are two modes of configuration a developer or user should consider :

1. Edge Peer
   A peer which may or may not be behind a firewall or NAT (i.e. Directly adressable or not). It is recommended that this class of peer should always be configured with TCP/IP enabled (both incoming/outgoing, multicast on), and HTTP enabled outgoing only, it should also use a relay, and a rendezvous. It is important to note that the JXTA platform automatically determines whether directs routes exists between peers, and will prefer such routes over relayed ones (hence the recommeded configuration)
2. Rendezvous/Relay Peer
   This class of peers is expected to provide infrastructure services and typically is directly reachable on the internet. It is recommended that this class of peer should always be configured with TCP/IP enabled (both incoming/outgoing, multicast on), and HTTP enabled incoming only, act as a relay, and rendezvous.

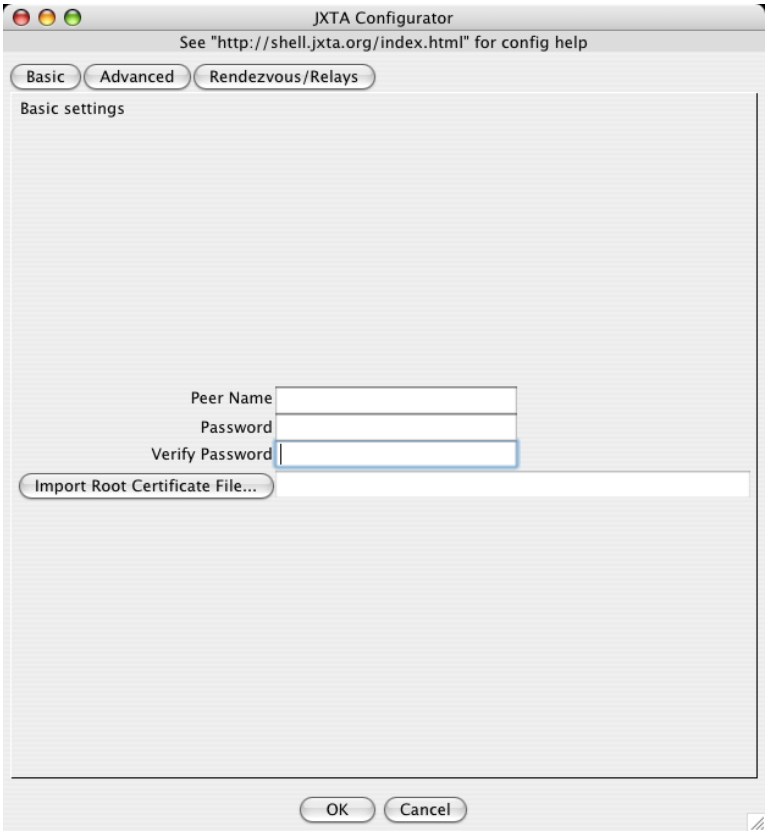
The first time a JXTA technology application is run, an auto-configuration tool (JXTA Configurator) is displayed to configure the JXTA platform for your network environment. This tool is used to specify configuration information for TCP/IP and HTTP, configure rendezvous and relay peers, and enter a user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

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

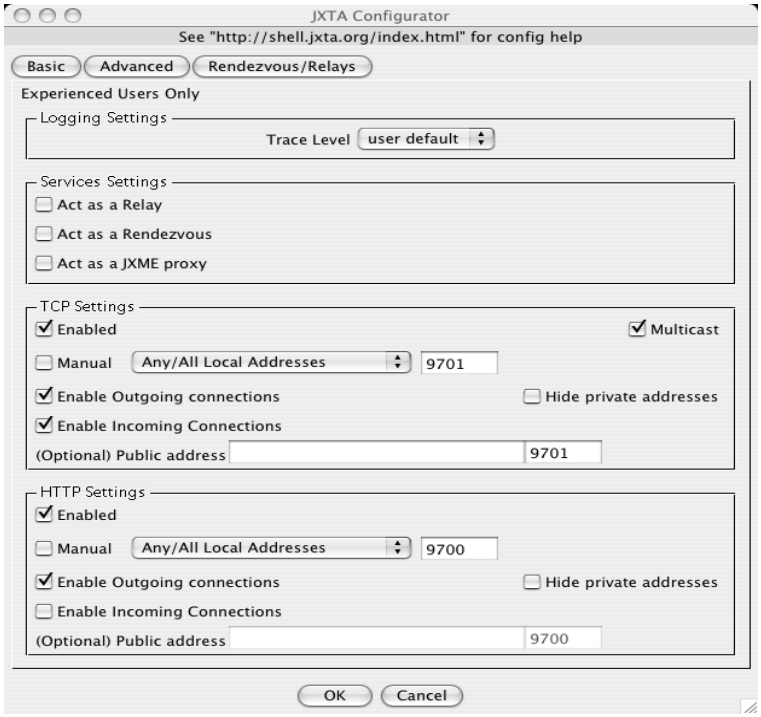
- Basic: You can use any string for your peer name. If your peer is located behind a firewall, you will also need to check the box "Use a proxy server" and enter your proxy server name and port number[9].
- Advanced: This panel is used to specify TCP and HTTP settings. Outgoing TCP connections should be enabled for most situations. If you are not behind a firewall or NAT, incoming TCP connections should also be enabled, and you do not need to use a relay. If you are behind a firewall or NAT, incoming connections should be disabled and a relay is needed in order to communicate.
- Rendezvous/Relay: Download the list of rendezvous and relay peers. If you are behind a firewall or NAT, select Use a Relay.
- Security: Enter a username and password.

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

9 Basic peer configuration



JXTA Configurator: Basic settings.



JXTA Configurator: Transport settings.

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

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

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

HelloWorld Example

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

0Example output: SimpleJxtaApp.

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

Hello World Example: SimpleJxtaApp

The code for this example begins on page 35. We define a single class, SimpleJxtaApp, with one class variable:

- `PeerGroup netPeerGroup` — our peer group (the default net peer group) and two methods:
- `static public void main()` — main routine; prints peer and peer group information
- `public void startJxta()` — initializes the JXTA platform and creates the net peer group

startJxta()

The startJxta() method uses a single call to instantiate the JXTA platform :

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

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

main()

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

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

Peer ID — the unique peer ID that was assigned to our JXTA peer when we ran the application :
```
System.out.println("  Peer ID = " +
                    netPeerGroup.getPeerID().toString());
```

After printing this information, the application calls the stopApp() method to stop the group services and then exits.
```
myapp.netPeerGroup.stopApp();
```

Running the Hello World Example

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

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

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

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:CP>
<jxta:CP type="jxta:PlatformConfig" xmlns:jxta="http://jxta.org">
        <PID>urn:jxta:uuid-59616261646162614A78746150325033B87CDE2608EA417AB843B23370A8E9C403</PID>
        <Name>e1</Name>
        <Desc> Platform Config Advertisement created by :
net.jxta.impl.peergroup.DefaultConfigurator</Desc>
        <Svc>
         <MCID>urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000505</MCID>
          <Parm type="jxta:PSEConfig" xmlns:jxta="http://jxta.org">
           <RootCert>
            <Certificate>
             cert omitted
            </Certificate>
           <EncryptedPrivateKey algorithm="RSA">
             cert omitted
           </EncryptedPrivateKey>
          </RootCert>
          </Parm>
        </Svc>
        <Svc>
         <MCID>urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000A05</MCID>
                <Parm>
                <jxta:TransportAdvertisement xmlns:jxta="http://jxta.org"
type="jxta:HTTPTransportAdvertisement">
                  <Protocol>http</Protocol>
                  <InterfaceAddress/>
                  <ConfigMode>auto</ConfigMode>
                  <Port>9700</Port>
                  <Proxy>myProxy.myDomain:8080</Proxy>
                  <ProxyOff/>
                  <ServerOff/>
                  <ClientOff/>
                 </jxta:TransportAdvertisement>
                 <isOff/>
                </Parm>
         </Svc>
        <Svc>
         <MCID>urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000E05</MCID>
         <Parm><isOff/></Parm>
        </Svc>
        <Svc>
         <MCID>urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000605</MCID>
          <Parm type="jxta:RdvConfig" config="client" xmlns:jxta="http://jxta.org"/>
        </Svc>
        <Svc>
         <MCID>urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000905</MCID>
          <Parm>
           <jxta:TransportAdvertisement xmlns:jxta="http://jxta.org"
type="jxta:TCPTransportAdvertisement">
            <Protocol>TCP</Protocol>
            <Port>9801</Port>
            <MulticastAddr>224.0.1.85</MulticastAddr>
            <MulticastPort>1234</MulticastPort>
            <MulticastSize>16384</MulticastSize>
            <ConfigMode>auto</ConfigMode>
           </jxta:TransportAdvertisement>
          </Parm>
        </Svc>
        <Svc>
        <MCID>urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000F05</MCID>
         <Parm>
          <isServer>false</isServer>
          <ServerMaximumClients/>
          <ServerLeaseInSeconds/>
          <isClient>false</isClient>
          <ClientMaximumServers/>
          <ClientLeaseInSeconds/>
          <ClientQueueSize>20</ClientQueueSize>
          <isOff/>
          </Parm>
        </Svc>
        <Svc>
         <MCID>urn:jxta:uuid-DEADBEEFDEAFBABAFEEDBABE0000000805</MCID>
         <Parm>
           <MessengerQueueSize>20</MessengerQueueSize>
         </Parm>
        </Svc>
</jxta:CP>
```

Source Code: SimpleJxtaApp

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

/**
 *  This is a simple example of how an application would start jxta
 */
public class SimpleJxtaApp {

    static PeerGroup netPeerGroup = null;

    public static void main(String args[]) {

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

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

Chapter 7: Programming with JXTA

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

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

1

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

Discovery Service

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

A JXTA peer can use the getLocalAdvertisements() method to retrieve advertisements that are in its local cache. If it wants to discover other advertisements, it uses getRemoteAdvertisements() to send a Discovery Query message to other peers. Discovery Query messages can be sent to a specific peer or propagated to the JXTA network. In the J2SE platform binding, Discovery Query messages not intended for a specific peer are propagated on the local subnet utilizing IP multicast and also sent to the peer's

rendezvous. Connection to the rendezvous peer occurs asynchronougly. If this peer has not yet connected to a rendezvous, the Discovery Query message will only be sent to the local subnet via multicast. Once the peer has connected to a rendezvous, the Discovery Query message will also be propagated to the rendezvous peer. A peer includes its own advertisement in the Discovery Query message, performing an announcement or automatic discovery mechanism.

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

The DiscoveryService is also used to publish advertisements. This is discussed in more detail in the "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"

DiscoveryDemo

This example uses the DisoveryListener interface to receive asynchronous notification of discovery events. [The code for this example begins on page 43 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 startJxta()` — initialize the JXTA platform
- `public void run()` — thread to send DiscoveryRequest messages
- `public void discoveryEvent(DiscoveryEvent ev)` — handle DiscoveryResponse messages that are received
- `static public void main()` — main routine

startJxta() method

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

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

Next, our discovery service is retrieved from our peer group, the netPeerGroup :

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

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

run() method

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

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

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

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

- `java.lang.string peerid` — ID of a peer to send query to; if null, propagate query request
- `int type` — DiscoveryService.PEER, DiscoveryService.GROUP, DiscoveryService.ADV

- `java.lang.string attribute` — attribute name to narrow discovery to
- `java.lang.string value` — value of attribute to narrow discovery to
- `int threshold` — the upper limit of responses from one peer

There are two main ways to send 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: DiscoveryDemo

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

public class DiscoveryDemo implements Runnable, DiscoveryListener {

    static PeerGroup netPeerGroup  = null;
    private DiscoveryService discovery;

   //start the JXTA platform
    private void startJxta() {
        try {
            netPeerGroup = PeerGroupFactory.newNetPeerGroup();
        } catch ( PeerGroupException e) {
            // could not instantiate the group, print the stack and exit
            System.out.println("fatal error : group creation failure");
            e.printStackTrace();
            System.exit(1);
        }
        // Get the discovery service from our peer group
        discovery = netPeerGroup.getDiscoveryService();
    }

    /**
     * This thread loops forever discovering peers
     * every minute, and displaying the results.
     */
    public void run() {
        try {
            // Add ourselves as a DiscoveryListener for Discovery events
            discovery.addDiscoveryListener(this);
            while (true) {
                System.out.println("Sending a Discovery Message");
                // look for any peer
                discovery.getRemoteAdvertisements(null,
                                  DiscoveryService.PEER,
                                  null, null, 5);
                // wait a bit before sending next discovery message
                try {
                    Thread.sleep(60 * 1000);
                } catch(Exception e) {}
            } //end while
        } catch(Exception e) {
            e.printStackTrace();
        }
    }

    /**
```

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

    public void discoveryEvent(DiscoveryEvent ev) {

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

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

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

Peer Group Discovery

Peer group discovery is very similar to the peer discovery in the previous example. The primary difference is that instead of sending DiscoveryRequest messages looking for peers, we send DiscoveryRequest messages looking for peer groups. Any DiscoveryResponse messages we receive will contain peer group advertisements rather than peer advertisements. In this example, however, after instantiating the JXTA platform we wait until we are connected to a rendezvous peer before sending Discovery Request messages. It would not be necessary to wait for a rendezvous connection if your application was running locally on a subnet and communicating with other peers on that subnet via multicast. However, in other configurations you might want to wait until a rendezvous connection is established before sending requests. For each DiscoveryResponse message received, the application prints the name of the peer sending the response (if it is known) as well as the name of each peer group that was discovered. Figure 7-2 shows example output when this application is run:

Example output: Peer group discovery example.

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

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

startJxta() method

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

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

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

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

run() method

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

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

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

discoveryEvent() method

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

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

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

Now, instead of receiving an enumeration of peer advertisements, we receive an enumeration peer group advertisements:

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

Source Code: GroupDiscoveryDemo

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


public class GroupDiscoveryDemo implements DiscoveryListener {

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

    /**
     *  Method to start the JXTA platform.
     *  Waits until a connection to rdv is established.
     */

    private void startJxta() {
        try {
            netPeerGroup = PeerGroupFactory.newNetPeerGroup();
        } catch ( PeerGroupException e) {
            // could not instantiate the group, print the stack and exit
            System.out.println("fatal error : group creation failure");
            e.printStackTrace();
            System.exit(1);
        }
        // Extract the discovery and rendezvous services
        // from our peer group
        discovery = netPeerGroup.getDiscoveryService();
        rdv = netPeerGroup.getRendezVousService();

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

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

```
     */

    public void run() {

        try {
            // Add ourselves as a DiscoveryListener for
            // DiscoveryResponse events
            discovery.addDiscoveryListener(this);
            while (true) {
                System.out.println("Sending a Discovery Message");
                // look for any peer group
                discovery.getRemoteAdvertisements(null,
                                                  DiscoveryService.GROUP,
                                                  null, null, 5);
                // wait a bit before sending next discovery message
                try {
                    Thread.sleep( 60 * 1000);
                } catch(Exception e) {}

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

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

    public void discoveryEvent(DiscoveryEvent ev) {

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

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

        if (en != null ) {
            while (en.hasMoreElements()) {
                adv = (PeerGroupAdvertisement) en.nextElement();
                System.out.println (" Peer Group = " + adv.getName());
            }
        }
    }
    static public void main(String args[]) {
```

```
        GroupDiscoveryDemo myapp  = new GroupDiscoveryDemo();
        myapp.startJxta();
        myapp.run();
    }

}
```

Creating Peer Groups and Publishing Advertisements

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

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

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

Figure 0 `./.jxta/cm/jxta-NetGroup` — local cache directory containing advertisements for the net peer group

Figure 1 `./.jxta/cm/1D5E451AF1B243C1AD39B9D331AE858C02` — cache directory for the new peer group

main()

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

startJxta()

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

Figure 0Instantiates the JXTA platform and creates the default net peer group :

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

Figure 0Extracts the discovery service from the peer group; this is used later to publish the new group advertisement :

```
discoSvc = myGroup.getDiscoveryService();
```

groupsInLocalCache()

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

Figure 0`int type` — DiscoveryService.PEER, DiscoveryService.GROUP, DiscoveryService.ADV

Figure 1`java.lang.string attribute` — attribute name to narrow discovery to

Figure 2`java.lang.string value` — value of attribute to narrow discovery to

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

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

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

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

createGroup()

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

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

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

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

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

We pass four arguments to newGroup():

Figure 0`PeerGroup ID gid` — the peer group ID of the group to be created; if null, a new peer group ID is generated

Figure 1`Advertisement implAdv` — the implementation advertisement

Figure 2`String name` — the name of the new group

Figure 3`String description` — a group description

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

> Note – Since the `myGroup`.newGroup() method publishes the new group for us, it is not necessary to explicitly call DiscoveryService.publish().
>
> After the group is created, we print the name of the group and its peer group ID.

The second part of this method publishes the new peer group advertisement remotely :

```
discoSvc.remotePublish(adv);
```

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

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

Source Code: PublishDemo

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

public class PublishDemo   {

    static PeerGroup myGroup = null;
    private DiscoveryService discoSvc;

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

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

    // print all peer groups found in the local cache
    private void groupsInLocalCache() {
        System.out.println("--- local cache (Peer Groups)  ---");
        try {
            PeerGroupAdvertisement adv = null;
            Enumeration en = discoSvc.getLocalAdvertisements(
                                       discoSvc.GROUP, null, null);
            if (en != null) {
                while (en.hasMoreElements()) {
                    adv = (PeerGroupAdvertisement) en.nextElement();
                    System.out.println(adv.getName() +
                                       ", group ID = " +
                                       adv.getPeerGroupID().toString());
                }
            }
        } catch (Exception e) {}
        System.out.println("--- end local cache ---");
    }
```

```
    // create and publish a new peer group
    private void createGroup() {
        PeerGroupAdvertisement adv;
        System.out.println("Creating a new group advertisement");
        try {
            // create a new all purpose peergroup.
            ModuleImplAdvertisement implAdv =
                myGroup.getAllPurposePeerGroupImplAdvertisement();
            PeerGroup pg = myGroup.newGroup(null, // Assign new group ID
                                        implAdv, // The implem. adv
                                        "PubTest",// The name
                                        "testing group adv"); // descr.

            // print the name of the group and the peer group ID
            adv = pg.getPeerGroupAdvertisement();
            PeerGroupID GID = adv.getPeerGroupID();
            System.out.println("  Group = " +adv.getName() +
                               "\n  Group ID = " + GID.toString());
        } catch (Exception eee) {
            System.out.println("Group creation failed with " +
                                                    eee.toString());
            return;
        }
        try {
            // publish this advertisement
            //(send out to other peers and rendezvous peer)
            discoSvc.remotePublish(adv);
            System.out.println("Group published successfully.");
        } catch (Exception e) {
            System.out.println("Error publishing group advertisement");
            e.printStackTrace();
            return;
        }
    }
}
```

Joining a Peer Group

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

shows example output when this application is run:

Example output: Creating and joining a peer group.

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

Joining peer group...
Successfully joined group JoinTest

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

Good Bye ....
```

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

Membership Service

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

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

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

Figure 0*Apply*

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

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

Figure 0*Join*

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

Figure 0*Resign*

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

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

main()

This method calls the remaining three class methods:

Figure 0startJxta() — to instantiate the JXTA platform and create the default net peer group

Figure 1createGroup() — to create and publish a new peer group

Figure 2joinGroup() — to join the new group

startJxta()

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

createGroup()

This method is almost identical to the createGroup() method in the previous example (see description on page 51). It is used to create a new peer group and publish its advertisement. The only significant change is that if the group is successfully created, this method returns the new PeerGroup. If there is an error creating the new peer group, this method returns null.

joinGroup()

This method is used to join the peer group that is passed as an argument :

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

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

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

This constructor takes three arguments:

Figure 0`PeerGroup peergroup` — the peer group context in which this AuthenticationCredential is created (i.e., the peer group that you want to join).

`java.lang.String method` — The authentication method which will be requested when the AuthenticationCredential is provided to the peer group MembershipService service.

Figure 0`Element IdentityInfo` — Optional additional information about the identity being requested, which is used by the authentication method. This information is passed to the authentication method during the apply operation of the MembershipService service.

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

Next, our example extracts the MembershipService from the peer group we want to join :

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

And uses the MembershipService.apply() method to apply for group membership :

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

The authentication credentials created earlier in the method are passed to the apply() method. Included in

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

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

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

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

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

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

Source Code: JoinDemo

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

public class JoinDemo   {

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

    public static void main(String args[]) {

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

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

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

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

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

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

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

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

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

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

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



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

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

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

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

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

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

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

                StringWriter out = new StringWriter();
                doc.sendToWriter(out);
                System.out.println(out.toString());
                out.close();
            } else
                System.out.println("Failure: unable to join group");
        } catch (Exception e){
            System.out.println("Failure in authentication.");
            e.printStackTrace();
        }
    }
}
```

Sending Messages Between Two Peers

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

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

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

0Example output: PipeListener.

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

0Example output: PipeExample.

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

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 61.

JXTA Pipe Service

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

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

Pipes are non-localized communication channels that are not bound to specific peers. This is a unique feature of JXTA pipes. The mechanism to resolve the location of pipes to a physical peer is done in a completely decentralized manner in JXTA via the JXTA Pipe Binding Protocol. The Pipe Binding Protocol does not rely on a centralized protocol such as

DNS (bind Hostname to IP) to bind a pipe advertisement (i.e., symbolic name) to an instance of a pipe on a physical peer (i.e., IP address). Instead, the resolver protocol uses a dynamic and adaptive search mechanism that attempts at all times to find the peers where an instance of that pipe is running.

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

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

PipeListener

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

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

We also define four instance fields:

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

main()

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

startJxta()

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

```
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 reading it in from the existing file `examplepipe.adv` :

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

The file `examplepipe.adv` must exist and it must be valid XML document containing a pipe advertisement, or an exception is raised by the JXTA platform. Both this application (which creates the input pipe) and the partner application (which creates the output pipe) read their pipe advertisement from the same file. The contents of the `examplepipe.adv` file are listed in on page 72.

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.

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

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

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:

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

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

pipeMsgEvent()

This method 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 PipeListener 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: PipeListener

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

import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.MessageElement;
import net.jxta.endpoint.Message.ElementIterator;
import net.jxta.exception.PeerGroupException;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.pipe.InputPipe;
import net.jxta.pipe.PipeMsgEvent;
import net.jxta.pipe.PipeMsgListener;
import net.jxta.pipe.PipeService;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.impl.endpoint.WireFormatMessage;
import net.jxta.impl.endpoint.WireFormatMessageFactory;
import net.jxta.util.CountingOutputStream;
import net.jxta.util.DevNullOutputStream;
/**
 * this application creates an instance of an input pipe,
 * and waits for msgs on the input pipe
 *
 */

public class PipeListener implements PipeMsgListener {

    static PeerGroup netPeerGroup = null;
    private final static String SenderMessage = "PipeListenerMsg";
    private PipeService pipe;
    private PipeAdvertisement pipeAdv;
    private InputPipe pipeIn = null;

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

        PipeListener myapp = new PipeListener();
        myapp.startJxta();
        myapp.run();
    }
    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"),
                        (MimeMediaType[]) null);
            System.out.println("Message Size :" +
                                serialed.getByteLength());
            while (it.hasNext()) {
                MessageElement el = (MessageElement) 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();
        }
    }

    /**
     * wait for msgs
     *
     */
    public void run() {

        try {
            // the following creates the inputpipe, and registers
            // “this” as the PipeMsgListener, when a message arrives
            // pipeMsgEvent is called
            System.out.println("Creating input pipe");
            pipeIn = pipe.createInputPipe(pipeAdv, this);
        } catch (Exception e) {
            return;
        }
        if (pipeIn == null) {
            System.out.println(" cannot open InputPipe");
            System.exit(-1);
        }
        System.out.println("Waiting for msgs on input pipe");
    }


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

            // uncomment the following line if you want to start the
            // app defined the NetPeerGroup Advertisement
            // (by default it's the shell) at which case you
```

```
            // must include jxtashell.jar in the classpath
            // in this case we want use jxta directly.
            // netPeerGroup.startApp(null);

        } 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);
        }

        pipe = netPeerGroup.getPipeService();
        System.out.println("Reading in pipexample.adv");
        try {
            FileInputStream is = new FileInputStream("pipexample.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);
        }
    }

    /**
     * By implementing PipeMsgListener, define this method to deal with
     * messages as they arrive
     */

    public void pipeMsgEvent(PipeMsgEvent event) {

        Message msg=null;
        try {
            // grab the message from the event
            msg = event.getMessage();
            if (msg == null) {
                return;
            }
            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 named SenderMessage
        MessageElement msgElement = msg.getMessageElement(null,
                                                          SenderMessage);
        // 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());
            }
        }

}
```

PipeExample

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

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

main()

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

run()

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

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

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

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

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

We then check if we are connected to a JXTA rendezvous peer :

```
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 PipeExample class to contain the element name:

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

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

```
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();
```

rendezvousEvent()

This method is called asynchronously whenever we receive a RendezvousEvent. This method is passed one argument:

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

We expect to receive a connection event (RDVCONNECT) when our peer connects to its rendezvous peer. Other possible events include disconnection events (RDVDISCONNECT), reconnection events (RDVRECONNECT), and rendezvous failure events (RDVFAILED).[10]

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

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

startJxta()

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

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

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

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

We then register a rendezvous listener :

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

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

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

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

The file `examplepipe.adv` must exist and it must be valid XML document containing a pipe advertisement, or an exception is raised by the JXTA platform. Recall that the application which creates the input pipe also reads its pipe advertisement from the same file. The contents of the `examplepipe.adv` file are listed in on page 72.

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();
```

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

Source Code: PipeExample

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

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

/**
 *  This exapmle illustrates how to use the OutputPipeListener interface
 *
 */
public class PipeExample implements
                              Runnable,
                              OutputPipeListener,
                              RendezvousListener {

    static PeerGroup netPeerGroup = null;
    private final static String SenderMessage = "PipeListenerMsg";
    private PipeService pipe;
    private DiscoveryService discovery;
    private PipeAdvertisement pipeAdv;
    private RendezVousService rendezvous;

    /**
     *  main
     *
     *@param  args  command line arguments
     */
    public static void main(String args[]) {
        PipeExample myapp = new PipeExample();
        myapp.startJxta();
        myapp.run();
    }

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

    public synchronized void run() {
        try {
            // create output pipe with asynchronously
```

```
            // Send out the first pipe resolve call
            System.out.println("Attempting to create a OutputPipe");
            pipe.createOutputPipe(pipeAdv, this);
            // send out a second pipe resolution after we connect
            // to a rendezvous
            if (!rendezvous.isConnectedToRendezVous()) {
                System.out.println("Waiting for Rendezvous Connection");
                try {
                    wait();
                    System.out.println("Connected to Rendezvous,
                                        attempting to create a OutputPipe");
                    pipe.createOutputPipe(pipeAdv, this);
                } catch (InterruptedException e) {
                    // got our notification
                }
            }
        } 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(" Got an output pipe event");
        OutputPipe op = event.getOutputPipe();
        Message msg = null;

        try {
            System.out.println("Sending message");
            msg = new Message();
            Date date = new Date(System.currentTimeMillis());
            StringMessageElement sme = new StringMessageElement(
                                SenderMessage, date.toString() , null);
            msg.addMessageElement(null, sme);
            op.send(msg);
        } catch (IOException e) {
            System.out.println("failed to send message");
            e.printStackTrace();
            System.exit(-1);
        }
        op.close();
        System.out.println("message sent");
    }

    /**
     *  rendezvousEvent the rendezvous event
     *
     *@param  event   rendezvousEvent
```

```
     */
    public synchronized void rendezvousEvent(RendezvousEvent event) {
        if (event.getType() == event.RDVCONNECT ||
            event.getType() == event.RDVRECONNECT ) {
            notify();
        }
    }

    /**
     *  Starts jxta, and get the pipe, and discovery service
     */
    private void startJxta() {
        try {
            // create, and Start the default jxta NetPeerGroup
            netPeerGroup = PeerGroupFactory.newNetPeerGroup();
            rendezvous = netPeerGroup.getRendezVousService();
            rendezvous.addListener(this);
            // uncomment the following line if you want to start
            // the app defined the NetPeerGroup Advertisement
            // (by default it's the shell)
            // in this case we want use jxta directly.
            // netPeerGroup.startApp(null);

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

        // get the pipe service, and discovery
        pipe = netPeerGroup.getPipeService();
        discovery = netPeerGroup.getDiscoveryService();
        System.out.println("Reading in pipexample.adv");
        try {
            FileInputStream is = new FileInputStream("pipexample.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);
        }
    }
}
```

Pipe Advertisement: `examplepipe.adv` file

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

0Pipe advertisement file, `examplepipe.adv`.

```
<!DOCTYPE jxta:PipeAdvertisement>

<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
      <Id>
       urn:jxta:uuid-
59616261646162614A757874614D504725184FBC4E5D498AA0919F662E400
28B04
      </Id>
      <Type>
       JxtaUnicast
      </Type>
      <Name>
       PipeExample
      </Name>
</jxta:PipeAdvertisement>
```

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

Using a JxtaBiDiPipe (A bidirectional reliable pipe)

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. It reads in pipe.adv which contains the pipe advertisement. After the pipe advertisement is read 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.

- `File pipe.adv` — The XML file containing the text representation of our pipe advertisement
- `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().

startJxta()

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

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

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

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.FileInputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.util.Date;
import net.jxta.credential.AuthenticationCredential;
import net.jxta.credential.Credential;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.MessageElement;
import net.jxta.endpoint.Messenger;
import net.jxta.endpoint.StringMessageElement;
import net.jxta.exception.PeerGroupException;
import net.jxta.membership.InteractiveAuthenticator;
import net.jxta.membership.MembershipService;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.util.JxtaBiDiPipe;
import net.jxta.util.JxtaServerPipe;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.impl.protocol.PlatformConfig;
import org.apache.log4j.Logger;

/**
 *  This example illustrates how to utilize the JxtaBiDiPipe Reads in
 *  pipe.adv and attempts to bind to a JxtaServerPipe
 */
public class JxtaServerPipeExample  {
    public static final int ITERATIONS = 100;
    private PeerGroup netPeerGroup = null;
    private PipeAdvertisement pipeAdv;
    private JxtaServerPipe serverPipe;
    private static final MimeMediaType MEDIA_TYPE = new
                                MimeMediaType("application/bin");
    private final static Logger LOG = Logger.getLogger(
                                  JxtaServerPipeExample.class.getName());
    private final static String SenderMessage = "pipe_tutorial";

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

        JxtaServerPipeExample eg = new JxtaServerPipeExample();
        eg.startJxta();
        System.out.println("Reading in pipe.adv");
        try {
            FileInputStream is = new FileInputStream("pipe.adv");
            eg.pipeAdv = (PipeAdvertisement)
```

```
                AdvertisementFactory.newAdvertisement(
                        MimeMediaType.XMLUTF8, is);
            is.close();
            eg.serverPipe = new JxtaServerPipe(eg.netPeerGroup,
                                               eg.pipeAdv);
            // we want to block until a connection is established
            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();
    }


    private void sendTestMessages(JxtaBiDiPipe pipe) {
        try {
            for (int i =0; i<ITERATIONS; i++) {
            Message msg = new Message();
            String data = "Message #"+i;
            msg.addMessageElement(SenderMessage,
                        new StringMessageElement(SenderMessage,
                                                 data,
                                                 null));
            System.out.println("Sending :"+data);
            pipe.sendMessage(msg);
            //Thread.sleep(100);
            }
        } catch (Exception ie) {
            ie.printStackTrace();
        }
    }

    /**
     * wait for msgs
     *
     */
    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 100 messages to the other end");
                    //Send a 100 messages
                    sendTestMessages(bipipe);
                }
            } catch (Exception e) {
                e.printStackTrace();
                return;
            }
        }
```

```
    }

    /**
     * Starts jxta
     *
     */
    private void startJxta() {
        try {
            System.setProperty("net.jxta.tls.principal", "server");
            System.setProperty("net.jxta.tls.password", "password");
            System.setProperty("JXTA_HOME",
            System.getProperty("JXTA_HOME", "server"));
            File home = new File(
                     System.getProperty("JXTA_HOME", "server"));
            if (!configured(home)) {
                createConfig(home, "JxtaServerPipeExample", true);
            }
            // create, and Start the default jxta NetPeerGroup
            netPeerGroup = PeerGroupFactory.newNetPeerGroup();
            JxtaBidiPipeExample.login(netPeerGroup, "server", "password");
            //netPeerGroup.startApp(null);
        } 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);
        }
    }

    /**
     *Returns a resource InputStream
     *
     *@param  resource         resource name
     *@return                  returns a resource InputStream
     *@exception  IOException  if an I/O error occurs
     */
    protected static InputStream getResourceInputStream(
                          String resource) throws IOException {
      ClassLoader cl = JxtaServerPipeExample.class.getClassLoade();
        return cl.getResourceAsStream(resource);
    }
    /**
     *Returns true if the node has been configured, otherwise false
     *
     *@param  home  node jxta home directory
     *@return       true if home/PlatformConfig exists
     */
    protected static boolean configured(File home) {
        File platformConfig = new File(home, "PlatformConfig");
        return platformConfig.exists();
    }
    /**
     *  Creates a PlatformConfig with peer name set to name
     *
     *@param  home  node jxta home directory
     *@param  name  node given name (can be hostname)
     */
    protected static void createConfig(File home,
```

```
                        String name, boolean server) {
        try {
            String fname = null;
            if (server) {
                fname = "ServerPlatformConfig.master";
            } else {
                fname = "PlatformConfig.master";
            }
            InputStream is = getResourceInputStream(fname);
            home.mkdirs();
            PlatformConfig platformConfig = (PlatformConfig)
             AdvertisementFactory.newAdvertisement(
                                    MimeMediaType.XMLUTF8, is);
            is.close();
            platformConfig.setName(name);
            File newConfig = new File(home, "PlatformConfig");
            OutputStream op = new FileOutputStream(newConfig);
            StructuredDocument doc = (StructuredDocument)
                 platformConfig.getDocument(MimeMediaType.XMLUTF8);
            doc.sendToStream(op);
            op.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
}
```

Example pipe advertisement: pipe.adv

An example pipe advertisement, saved to the file `pipe.adv`, 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. pipe.adv contains the pipe advertisement which both ends binds to.

- `File pipe.adv` — The XML file containing the text representation of our pipe advertisement
- `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
- `PipeAdvertisement pipeAdv` — the pipe advertisement used in this example
- `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.

```
FileInputStream is = new FileInputStream("pipe.adv");
eg.pipeAdv = (PipeAdvertisement) AdvertisementFactory.
                  newAdvertisement(MimeMediaType.XMLUTF8, is);
is.close();
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);
```

startJxta()

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

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

It then gets establishes it's credentials within the netpeerGroup :

```
login(netPeerGroup, "principal", "password");
```

Source Code: JxtaBidiPipeExample

```
import java.io.File;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.util.Date;
import net.jxta.credential.AuthenticationCredential;
import net.jxta.credential.Credential;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.endpoint.Message;
import net.jxta.endpoint.MessageElement;
import net.jxta.endpoint.Messenger;
import net.jxta.endpoint.StringMessageElement;
import net.jxta.exception.PeerGroupException;
import net.jxta.impl.membership.pse.StringAuthenticator;
import net.jxta.membership.InteractiveAuthenticator;
import net.jxta.membership.MembershipService;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.pipe.PipeMsgEvent;
import net.jxta.pipe.PipeMsgListener;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.util.JxtaBiDiPipe;
import net.jxta.rendezvous.RendezvousEvent;
import net.jxta.rendezvous.RendezvousListener;
import net.jxta.rendezvous.RendezVousService;
import org.apache.log4j.Level;
import org.apache.log4j.Logger;

/**
 *  This example illustrates how to utilize the JxtaBiDiPipe Reads in
 *  pipe.adv and attempts to bind to a JxtaServerPipe
 */

public class JxtaBidiPipeExample implements PipeMsgListener, RendezvousListener {

    private PeerGroup netPeerGroup = null;
    private PipeAdvertisement pipeAdv;
    private JxtaBiDiPipe pipe;
    private RendezVousService rendezvous;
    private final static String SenderMessage = "pipe_tutorial";
    private final static String completeLock = "completeLock";
    private int count = 0;

    private final static Logger LOG = Logger.getLogger
(JxtaBidiPipeExample.class.getName());

    /**
     *  Starts jxta
     */
    private void startJxta() {
        try {
            System.setProperty("net.jxta.tls.principal", "client");
```

```
            System.setProperty("net.jxta.tls.password", "password");
            System.setProperty("JXTA_HOME",
                        System.getProperty("JXTA_HOME", "client"));
            File home = new File(System.getProperty("JXTA_HOME",
                                                    "client"));
            if (!JxtaServerPipeExample.configured(home)) {
                JxtaServerPipeExample.createConfig(home,
                                "JxtaBidiPipeExample", false);
            }

            // create, and Start the default jxta NetPeerGroup
            netPeerGroup = PeerGroupFactory.newNetPeerGroup();
            rendezvous = netPeerGroup.getRendezVousService();
            login(netPeerGroup, "client", "password");
            netPeerGroup.startApp(null);
        } 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);
        }
    }

    public static void login(PeerGroup group,
                             String principal, String password) {
        try {
            StringAuthenticator auth = null;
            MembershipService membership = group.getMembershipService();
            Credential cred = membership.getDefaultCredential();
            if (cred == null) {
                AuthenticationCredential authCred = new
                         AuthenticationCredential(group,
                            "StringAuthentication", null);
                try {
                    auth = (StringAuthenticator)
                                membership.apply(authCred);
                } catch(Exception failed) {
                    ;
                }

                if (auth != null) {
                    auth.setAuth1_KeyStorePassword(
                         password.toCharArray());
                    auth.setAuth2Identity(group.getPeerID());
                    auth.setAuth3_IdentityPassword(
                                principal.toCharArray());
                    if (auth.isReadyForJoin()) {
                        membership.join(auth);
                    }
                }
            }

            cred = membership.getDefaultCredential();
            if (null == cred) {
                AuthenticationCredential authCred = new
                   AuthenticationCredential(group,
                               "InteractiveAuthentication", null);
                InteractiveAuthenticator iAuth =
```

```
                    (InteractiveAuthenticator) membership.apply(
                                                            authCred);
                if (iAuth.interact() && iAuth.isReadyForJoin()) {
                    membership.join(iAuth);
                }
            }
        } catch(Throwable e) {
            // make sure output buffering doesn't wreck console display.
            System.out.flush();
            System.err.println("Uncaught Throwable caught by 'main':");
            e.printStackTrace();
            System.exit(1);
        } finally {
            System.err.flush();
            System.out.flush();
        }
    }
    /**
     *  when we get a message, print out the message on the console
     *
     *@param  event  message event
     */
    public void pipeMsgEvent(PipeMsgEvent event) {

        Message msg = null;
        try {
            // grab the message from the event
            msg = event.getMessage();
            if (msg == null) {
                if (LOG.isEnabledFor(Level.DEBUG)) {
                    LOG.debug("Received an empty message, returning");
                }
                return;
            }
            if (LOG.isEnabledFor(Level.DEBUG)) {
                LOG.debug("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 {
                Date date = new Date(System.currentTimeMillis());
                System.out.println("Message  :"+ msgElement.toString());
                count ++;
            }
            if (count >= JxtaServerPipeExample.ITERATIONS) {
                synchronized(completeLock) {
                    completeLock.notify();
                }
            }
        } catch (Exception e) {
            if (LOG.isEnabledFor(Level.DEBUG)) {
                LOG.debug(e);
            }
            return;
```

```
    }
}
/**
 * rendezvousEvent the rendezvous event
 *
 *@param  event   rendezvousEvent
 */
public synchronized void rendezvousEvent(RendezvousEvent event) {
    if (event.getType() == event.RDVCONNECT ||
        event.getType() == event.RDVRECONNECT ) {
        notify();
    }
}
/**
 * awaits a rendezvous connection
 */
private synchronized void waitForRendezvousConncection() {
    if (!rendezvous.isConnectedToRendezVous()) {
        System.out.println("Waiting for Rendezvous Connection");
        try {
            wait();
            System.out.println("Connected to Rendezvous");
        } catch (InterruptedException e) {
            // got our notification
        }
    }
}

private void waitUntilCompleted() {
    try {
        synchronized(completeLock) {
            completeLock.wait();
        }
        System.out.println("Done.");
    } catch (InterruptedException e) {
        System.out.println("Interrupted.");
    }
}

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

    JxtaBidiPipeExample eg = new JxtaBidiPipeExample();
    eg.startJxta();
    System.out.println("reading in pipe.adv");
    try {
        FileInputStream is = new FileInputStream("pipe.adv");
        eg.pipeAdv = (PipeAdvertisement)
                     AdvertisementFactory.newAdvertisement(
                             MimeMediaType.XMLUTF8, is);
        is.close();
        System.out.println("creating the BiDi pipe");
        eg.pipe = new JxtaBiDiPipe();
        eg.pipe.setReliable(true);
```

```
            eg.waitForRendezvousConncection();
            System.out.println("Attempting to establish a connection");
            eg.pipe.connect(eg.netPeerGroup,
                            null,
                            eg.pipeAdv,
                            180000,
                            // register as a message listener
                            eg);
            //at this point we need to keep references around
            //until data xchange is complete
            eg.waitUntilCompleted();
            System.exit(0);
        } catch (Exception e) {
            System.out.println("failed to bind the JxtaBiDiPipe due
                                    to the following exception");
            e.printStackTrace();
            System.exit(-1);
        }
    }

}
```

Using JxtaSockets (bidirectional reliable pipes with java.net.Socket interface)

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

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

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: □□□□□□□□□□□□□□□□□□.

```
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: □□□□□□□□□□□□□□□□□□□□□□□□□□.

```
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

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

- `File socket.adv` — The XML file containing the text representation of our pipe advertisement

We also 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().

startJxta()

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

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

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 the content of the socket.adv file then awaits data from the remote side

Source Code: JxtaServerSocketExample

```
import java.io.File;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import java.io.FileInputStream;
import java.net.Socket;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.exception.PeerGroupException;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.socket.JxtaServerSocket;
import net.jxta.protocol.PipeAdvertisement;

/**
 *  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 Once a connection is established, it sends
 *  the content of socket.adv and reads data from the remote side.
 *
 */

public class JxtaServerSocketExample {

    private transient PeerGroup netPeerGroup = null;
    private transient PipeAdvertisement pipeAdv;
    private transient JxtaServerSocket serverSocket;

    /**
     * Sends data over socket
     *
     *@param  socket  the socket
     */
    private void sendAndReceiveData(Socket socket) {
        try {
            // get the socket output stream
            OutputStream out = socket.getOutputStream();
            // read a file into a buffer
            File file = new File("socket.adv");
            FileInputStream is = new FileInputStream(file);
            int size = 4096;
            byte[] buf = new byte[size];
            int read = is.read(buf, 0, size);

            // send some bytes over the socket (the socket adv is used,
            // but that could be anything. It's just a handshake.)
            out.write(buf, 0, read);
            out.flush();
            System.out.println(read + " bytes sent");
            InputStream in = socket.getInputStream();

            // this call should block until bits are avail.
            long total = 0;
            long start = System.currentTimeMillis();
            while (true) {
```

```
                read = in.read(buf, 0, size);
                if (read < 1) {
                    break;
                }
                total += read;
                //System.out.print(".");
                //System.out.flush();
            }
            System.out.println("");
            long elapsed = System.currentTimeMillis() - start;
            System.out.println("EOT. Received " + total + " bytes in " +
                               elapsed + " ms. Throughput = " +
                     ((total * 8000) / (1024 * elapsed)) + " Kbit/s.");
            socket.close();
            System.out.println("Closed connection. Ready for next
                                                connection.");
        } catch (IOException ie) {
            ie.printStackTrace();
        }
    }

    /**
     *  wait for data
     */
    public void run() {

        System.out.println("starting ServerSocket");
        while (true) {
            try {
                System.out.println("Calling accept");
                Socket socket = serverSocket.accept();
                // set reliable
                if (socket != null) {
                    System.out.println("socket created");
                    sendAndReceiveData(socket);
                }
            } catch (Exception e) {
                e.printStackTrace();
            }
        }
    }

    /**
     *  Starts jxta
     */
    private void startJxta() {
        try {
            System.setProperty("net.jxta.tls.principal", "server");
            System.setProperty("net.jxta.tls.password", "password");
            System.setProperty("JXTA_HOME", System.getProperty("JXTA_HOME",
"server"));
            File home = new File(System.getProperty("JXTA_HOME", "server"));
            if (!JxtaSocketExample.configured(home)) {
                JxtaSocketExample.createConfig(home, "JxtaServerSocketExample",
true);
            }

            // create, and Start the default jxta NetPeerGroup
```

```
            netPeerGroup = PeerGroupFactory.newNetPeerGroup();
            //JxtaSocketExample.login(netPeerGroup, "server", "password");
        } 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);
        }
    }

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

        JxtaServerSocketExample socEx = new JxtaServerSocketExample();
        socEx.startJxta();
        System.out.println("Reading in socket.adv");
        try {
            FileInputStream is = new FileInputStream("socket.adv");
            socEx.pipeAdv = (PipeAdvertisement) AdvertisementFactory.newAdvertisement
(MimeMediaType.XMLUTF8, is);
            is.close();
            socEx.serverSocket = new JxtaServerSocket(socEx.netPeerGroup,
socEx.pipeAdv, 10);
            // block until a connection is available
            socEx.serverSocket.setSoTimeout(0);
        } catch (Exception e) {
            System.out.println("failed to read/parse pipe advertisement");
            e.printStackTrace();
            System.exit(-1);
        }
        socEx.run();
    }
}
```

Example pipe advertisement: socket.adv

An example pipe advertisement, saved to the file `socket.adv`, is listed below:

```
<!DOCTYPE jxta:PipeAdvertisement>

<jxta:PipeAdvertisement xmlns:jxta="http://jxta.org">
        <Id>
urn:jxta:uuid-59616261646162614E5047205032503393B5C2F6CA7A41FBB0F890173088E79404
        </Id>
        <Type>
                JxtaUnicast
        </Type>
        <Name>
                socket tutorial
        </Name>
</jxta:PipeAdvertisement>
```

JxtaSocketExample

This application creates JxtaSocket and attempts to connect to JxtaServerSocket. socket.adv contains the pipe advertisement which both ends binds to.

- `File socket.adv` — The XML file containing the text representation of our pipe advertisement
- We also define the following instance fields:
- `PeerGroup netPeerGroup` — our peer group, the default net peer group
- `PipeAdvertisement pipeAdv` — the pipe advertisement used in this example
- `JxtaSocket socket` — the JxtaSocket used to connect to JxtaServerSocket

main()

This method creates a new JxtaSocketExample object, calls startJxta() 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.

```
JxtaSocketExample socEx = new JxtaSocketExample();
     System.out.println("Starting JXTA");
     socEx.startJxta();
     System.out.println("reading in socket.adv");
     FileInputStream is = new FileInputStream("socket.adv");
     socEx.pipeAdv = (PipeAdvertisement)
         AdvertisementFactory.newAdvertisement(
                                  MimeMediaType.XMLUTF8, is);
     is.close();
     // run it once
     socEx.run();
     // run it again, to exclude any object
     // initialization overhead
     socEx.run();
```

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: JxtaSocketExample

```
import java.io.File;
import java.io.FileInputStream;
import java.io.FileOutputStream;
import java.io.InputStream;
import java.io.IOException;
import java.io.OutputStream;
import net.jxta.credential.AuthenticationCredential;
import net.jxta.credential.Credential;
import net.jxta.document.AdvertisementFactory;
import net.jxta.document.MimeMediaType;
import net.jxta.document.StructuredDocument;
import net.jxta.exception.PeerGroupException;
import net.jxta.impl.membership.pse.StringAuthenticator;
import net.jxta.impl.protocol.PlatformConfig;
import net.jxta.membership.InteractiveAuthenticator;
import net.jxta.membership.MembershipService;
import net.jxta.peergroup.PeerGroup;
import net.jxta.peergroup.PeerGroupFactory;
import net.jxta.protocol.PipeAdvertisement;
import net.jxta.socket.JxtaSocket;

/**
 *  This tutorial illustrates the use JxtaSocket. It attempts to bind a
 *  JxtaSocket to an instance of JxtaServerSocket bound socket.adv. Once a
 *  connection is established, it reads in expected data from the remote
 *  side, and then sends 1824 64K chunks and measures data rate achieved
 *
 */

public class JxtaSocketExample {

    private transient PeerGroup netPeerGroup = null;
    private transient PipeAdvertisement pipeAdv;
    private transient JxtaSocket socket;
    // number of iterations to send the payload
    private static int ITERATIONS = 1824;
    // payload size
    private static int payloadSize = 64 * 1024;


    /**
     *  Interact with the server.
     *
     *@exception  IOException  if an io exception occurs
     */
    public void run() throws IOException {

        int bufsize = 1024;
        System.out.println("Connecting to the server");
        socket = new JxtaSocket(netPeerGroup,
                                //no specific peerid
                                null,
                                pipeAdv,
                                //general TO: 30 seconds
                                30000,
```

```
                                // reliable connection
                                true);

        // Set buffer size to payload size
        socket.setOutputStreamBufferSize(65536);

        // The server initiates communication by sending a small data packet
        // and then awaits data from the client
        System.out.println("Reading in data");
        InputStream in = socket.getInputStream();
        byte[] inbuf = new byte[bufsize];
        int read = in.read(inbuf, 0, bufsize);
        System.out.println("received " + read + " bytes");

        // Server is awaiting this data
        // Send data and time it.
        System.out.println("Sending back " + payloadSize +
                    " * " + ITERATIONS + " bytes");
        OutputStream out = socket.getOutputStream();
        byte[] payload = new byte[payloadSize];
        long t0 = System.currentTimeMillis();
        for (int i = 0; i < ITERATIONS; i++) {
            out.write(payload, 0, payloadSize);
        }
        out.flush();
        // include close in timing since it may need to flush the
        // tail end of the stream.
        socket.close();
        long t1 = System.currentTimeMillis();
        System.out.println("Completed in :" + (t1 - t0) + " msec");
        System.out.println("Data Rate :" +
            ((long) 64 * ITERATIONS * 8000) / (t1 - t0) + " Kbit/sec");
    }


    /**
     * Starts the NetPeerGroup, and logs in
     *
     *@exception  PeerGroupException  if a PeerGroupException occurs
     */
    private void startJxta() throws PeerGroupException {
       System.setProperty("net.jxta.tls.principal", "client");
       System.setProperty("net.jxta.tls.password", "password");
       System.setProperty("JXTA_HOME",
            System.getProperty("JXTA_HOME", "client"));
       File home = new File(System.getProperty("JXTA_HOME", "client"));
       if (!configured(home)) {
           createConfig(home, "JxtaSocketExample", false);
       }

        // create, and Start the default jxta NetPeerGroup
        netPeerGroup = PeerGroupFactory.newNetPeerGroup();

    }


    /**
     *  Establishes credentials with the specified peer group
```

```
     *
     *@param  group      PeerGroup
     *@param  principal  Principal
     *@param  password   password
     */
    public static void login(PeerGroup group, String principal, String password) {
        try {
            StringAuthenticator auth = null;
            MembershipService membership = group.getMembershipService();
            Credential cred = membership.getDefaultCredential();
            if (cred == null) {
                AuthenticationCredential authCred = new AuthenticationCredential
(group, "StringAuthentication", null);
                try {
                    auth = (StringAuthenticator) membership.apply(authCred);
                } catch (Exception failed) {
                    ;
                }

                if (auth != null) {
                    auth.setAuth1_KeyStorePassword(password.toCharArray());
                    auth.setAuth2Identity(group.getPeerID());
                    auth.setAuth3_IdentityPassword(principal.toCharArray());
                    if (auth.isReadyForJoin()) {
                        membership.join(auth);
                    }
                }
            }

            cred = membership.getDefaultCredential();
            if (null == cred) {
                AuthenticationCredential authCred = new
                 AuthenticationCredential(group,
                                          "InteractiveAuthentication", null);

                InteractiveAuthenticator iAuth = (InteractiveAuthenticator)
                                          membership.apply(authCred);

                if (iAuth.interact() && iAuth.isReadyForJoin()) {
                    membership.join(iAuth);
                }
            }
        } catch (Throwable e) {
            System.out.flush();
            // make sure output buffering doesn't wreck console display.
            System.err.println("Uncaught Throwable caught by 'main':");
            e.printStackTrace();
            System.exit(1);
            // make note that we abended
        }
        finally {
            System.err.flush();
            System.out.flush();
        }
    }

    /**
     *  returns a resource InputStream
```

```
     *
     *@param  resource         resource name
     *@return                  returns a resource InputStream
     *@exception  IOException  if an I/O error occurs
     */
    protected static InputStream getResourceInputStream(String resource) throws
IOException {
        ClassLoader cl = JxtaSocketExample.class.getClassLoader();
        return cl.getResourceAsStream(resource);
    }
    /**
     *  Returns true if the node has been configured, otherwise false
     *
     *@param  home  node jxta home directory
     *@return       true if home/PlatformConfig exists
     */
    protected static boolean configured(File home) {
        File platformConfig = new File(home, "PlatformConfig");
        return platformConfig.exists();
    }
    /**
     *  Creates a PlatformConfig with peer name set to name
     *
     *@param  home  node jxta home directory
     *@param  name  node given name (can be hostname)
     */
    protected static void createConfig(File home, String name,
                                                    boolean server) {
        try {
            String fname = null;
            if (server) {
                fname = "ServerPlatformConfig.master";
            } else {
                fname = "PlatformConfig.master";
            }
            InputStream is = getResourceInputStream(fname);
            home.mkdirs();
            PlatformConfig platformConfig = (PlatformConfig)
AdvertisementFactory.newAdvertisement(MimeMediaType.XMLUTF8, is);
            is.close();
            platformConfig.setName(name);
            File newConfig = new File(home, "PlatformConfig");
            OutputStream op = new FileOutputStream(newConfig);
            StructuredDocument doc = (StructuredDocument) platformConfig.getDocument
(MimeMediaType.XMLUTF8);
            doc.sendToStream(op);
            op.close();
        } catch (IOException e) {
            e.printStackTrace();
        }
    }
    /**
     *  main
     *
     *@param  args  none recognized.
     */
    public static void main(String args[]) {
```

```
        try {
            JxtaSocketExample socEx = new JxtaSocketExample();
            System.out.println("Starting JXTA");
            socEx.startJxta();
            System.out.println("reading in socket.adv");
            FileInputStream is = new FileInputStream("socket.adv");
            socEx.pipeAdv = (PipeAdvertisement) AdvertisementFactory.newAdvertisement
(MimeMediaType.XMLUTF8, is);
            is.close();
            // run it once
            socEx.run();
            // run it again, to exclude any object initialization overhead
            socEx.run();
        } catch (Throwable e) {
            System.out.println("failed : " + e);
            e.printStackTrace();
            System.exit(-1);
        }
        System.exit(0);
    }
}
```

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 ....
```

1

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

Server

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

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

- `String SERVICE` — the name of the service we create and advertise
- `String TAG` — the message element name, or tag, which we are expecting in any message we receive; the client application *must* use this same tag.
- `String NAMESPACE` — the namespace used by the message element; the client application *must* use this same space.
- `String FILENAME` — the name of the file that contains our pipe advertisement. (This file must exist and contain a valid pipe advertisement in order for our application to run correctly.)
- We also define several instance fields:
- `PeerGroup group` — our peer group, the default net peer group
- `DiscoveryService discoSvc` — the discovery service; used to publish our new service
- `PipeService pipeSvc` — the pipe service; used to create our input pipe and read messages
- `InputPipe myPipe` — the pipe used to receive messages

main()

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

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

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

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

```
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: Server

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

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.getAdver
tisementType());

            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: `pipeserver.adv` file

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

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

<!DOCTYPE jxta:PipeAdvertisement>

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

Client

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

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

- `String SERVICE` — the name of the service we are looking for (advertised by Server)
- `String TAG` — the message element name, or tag, which we include in any message we send; the Server application *must* use this same tag.
- `String NAMESPACE` — the namespace used by the message element; the client application *must* use this same space.
- We also define several instance fields:
- `PeerGroup netPeerGroup` — our peer group, the default net peer group
- `DiscoveryService discoSvc` — the discovery service; used to find the service
- `PipeService pipeSvc` — the pipe service; used to create our output pipe and send messages

main()

This method 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: Client

```
1. 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");
            }
        }
    }
```

Creating a Secure 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:

0Example 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>
                                SecurePeerGroups: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: SecurePeerGroups

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.

- Password: RULE

main()

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

The constructor method SecurePeerGroup()

This method creates and joins the secure peer group, and then prints the peer group's advertisement. More specifically, this method:

- Instantiates the JXTA platform and creates the default netPeerGroup by calling the startJxta() method
- 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.fromURL(new URL("urn","",
    "jxta:uuid-"+"DeadBeefDeafBabaFeedBabe00000001" +"04" +"06" ) );
```

```
passwdMembershipPeerGroupModuleImplAdv.setModuleSpecID(
    (ModuleSpecID) passwdGrpModSpecID);
```

CreatePasswdMembershipServiceModuleImplAdv()

This method works like the previous one: it takes a generic advertisement and uses it to create a customized one.

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 service
       </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: SecurePeerGroup

```
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 SecurePeerGroup  {

    private PeerGroup  myNetPeerGroup=null,
satellaPeerGroup=null,discoveredSatellaPeerGroup=null;
    private static PeerGroupID satellaPeerGroupID;
    private final static String GROUPID = "jxta:uuid-
4d6172676572696e204272756e6f202002";


    /** Creates new RootWS */
    public SecurePeerGroup() {
        // Starts the JXTA Platform
        myNetPeerGroup=this.startJxta();
        if (myNetPeerGroup!=null) {
            System.out.println("JXTA platform Started ...");
        } else {
            System.err.println(" JXTA platform has failed to start:
                                    myNetPeerGroup is null");
            System.exit(1);
        }
        //Generate the parameters:
        // login, passwd, peer group name and peer group id
```

```
        // for creating the Peer Group
        String login="SecurePeerGroups";
        String passwd="RULE";
        String groupName="SatellaGroup";
        // and finally peer group id
        // the peer group id is constant so that the same peer group is
        //recreated each time.
        try {
            satellaPeerGroupID =
                (PeerGroupID) net.jxta.id.IDFactory.fromURL(
                        new java.net.URL("urn","",GROUPID));
        } catch (java.net.MalformedURLException e) {
            System.err.println(" Can't create satellaPeerGroupID:
                      MalformedURLException") ;
            System.exit(1);
        } catch (java.net.UnknownServiceException e) {
            System.err.println(" Can't create satellaPeerGroupID:
                                  UnknownServiceException ") ;
            System.exit(1);
        }

        // create The Passwd Authenticated Peer Group
        satellaPeerGroup =this.createPeerGroup(
                         myNetPeerGroup,groupName,login,passwd);

        // join the satellaPeerGroup
        if (satellaPeerGroup!=null) {
            System.out.println(" Peer Group Created ...");
            discoveredSatellaPeerGroup=this.discoverPeerGroup(
                           myNetPeerGroup,satellaPeerGroupID);
            if (discoveredSatellaPeerGroup!=null) {
                System.out.println(" Peer Group Found ...");
                this.joinPeerGroup(discoveredSatellaPeerGroup,
                                       login, passwd);
            }
        }
        System.out.println(" Peer Group Joined ...");
        // Print the Peer Group Adverstisement on sdt out.
        this.printXmlAdvertisement("XML Advertisement for
                         Peer Group Advertisement",
                         satellaPeerGroup.getPeerGroupAdvertisement() );
    }

    private PeerGroup createPeerGroup(PeerGroup rootPeerGroup,
                        String groupName, String login, String passwd ) {
        // create the Peer Group by doing the following:
        // - Create a Peer Group Module Implementation Advertisement and publish it
        // - Create a Peer Group Adv and publish it
        // - Create a Peer Group from the Peer Group Adv and return this object
        PeerGroup satellaPeerGroup=null;
        PeerGroupAdvertisement satellaPeerGroupAdvertisement;

        // Create the PeerGroup Module Implementation Adv
        ModuleImplAdvertisement passwdMembershipModuleImplAdv ;
        passwdMembershipModuleImplAdv=this.createPasswdMembershipPeerGr
oupModuleImplAdv(rootPeerGroup);
        // 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 SecurePeerGroup,
        // 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.getAllPurposePeerG
roupImplAdvertisement();
        } 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.createPasswdM
embershipServiceModuleImplAdv
(allPurposePeerGroupMemershipServiceModuleImplAdv);
                //Remove the All purpose Membership Service
implementation
                allPurposePeerGroupServicesHashtable.remove
(allPurposePeerGroupServiceID);
                // And Replace it by the Passwd Membership Service
                // Implementation
                allPurposePeerGroupServicesHashtable.put(
                          PeerGroup.membershipClassID,
                          passwdMembershipServiceModuleImplAdv);
                membershipServiceFound=true;
                // Now the Service Advertisements are complete. Let's
                // update the passwdMembershipPeerGroupModuleImplAdv by
                // Updating its param
                passwdMembershipPeerGroupModuleImplAdv.setParam
((Element) passwdMembershipPeerGroupParamAdv.getDocument(new
MimeMediaType("text/xml")));
                // Update its Spec ID This comes from the
                // Instant P2P PeerGroupManager Code (Thanks !!!!)
                if
```

```
(!passwdMembershipPeerGroupModuleImplAdv.getModuleSpecID().equals
(PeerGroup.allPurposePeerGroupSpecID)) {
                    passwdMembershipPeerGroupModuleImplAdv.setModuleSpe
cID(IDFactory.newModuleSpecID
(passwdMembershipPeerGroupModuleImplAdv.getModuleSpecID().getBaseClass
()));
                } else {
                    ID passwdGrpModSpecID= ID.nullID;
                    try {
                        passwdGrpModSpecID = IDFactory.fromURL(new URL(
                         "urn",
                         "",
                         "jxta:uuid-"+
                       "DeadBeefDeafBabaFeedBabe00000001" +"04" +"06"));
                    } catch (java.net.MalformedURLException e) {}
                    catch (java.net.UnknownServiceException ee) {}
                    passwdMembershipPeerGroupModuleImplAdv.
                     setModuleSpecID((ModuleSpecID) passwdGrpModSpecID);
                } //End Else
                membershipServiceFound=true;
            } //end if (allPurposePeerGroupServiceID.
              // equals(PeerGroup.membershipClassID))
        }//end While

        return passwdMembershipPeerGroupModuleImplAdv;
    }

    private  ModuleImplAdvertisement
createPasswdMembershipServiceModuleImplAdv(ModuleImplAdvertisement
allPurposePeerGroupMemershipServiceModuleImplAdv) {
        //Create a new ModuleImplAdvertisement for the
        // Membership Service
        ModuleImplAdvertisement passwdMembershipServiceModuleImplAdv =
(ModuleImplAdvertisement) AdvertisementFactory.newAdvertisement
(ModuleImplAdvertisement.getAdvertisementType());
        passwdMembershipServiceModuleImplAdv.setModuleSpecID
(PasswdMembershipService.passwordMembershipSpecID);
        passwdMembershipServiceModuleImplAdv.setCode
(PasswdMembershipService.class.getName());
        passwdMembershipServiceModuleImplAdv.setDescription(" Module
Impl Advertisement for the PasswdMembership Service");
        passwdMembershipServiceModuleImplAdv.setCompat
(allPurposePeerGroupMemershipServiceModuleImplAdv.getCompat());
        passwdMembershipServiceModuleImplAdv.setUri
(allPurposePeerGroupMemershipServiceModuleImplAdv.getUri());
        passwdMembershipServiceModuleImplAdv.setProvider
(allPurposePeerGroupMemershipServiceModuleImplAdv.getProvider());
        return passwdMembershipServiceModuleImplAdv;
    }

    private PeerGroup discoverPeerGroup(PeerGroup myNetPeerGroup,
                                        PeerGroupID satellaPeerGroupID) {
        // First discover the peer group
        // In most cases we should use discovery listeners so that
        // we can do the discovery assynchroneously.
        // Here I won't, for increased simplicity and because
        // The Peer Group Advertisement is in the local cache for sure
        PeerGroup satellaPeerGroup;
```

```
        DiscoveryService myNetPeerGroupDiscoveryService=null;
        if (myNetPeerGroup!=null) {
            myNetPeerGroupDiscoveryService =
                        myNetPeerGroup.getDiscoveryService();
        } else {
            System.err.println("Can't join  Peer Group since
                                            its parent is null");
            System.exit(1);
        }
        boolean isGroupFound=false;
        Enumeration localPeerGroupAdvertisementEnumeration=null;
        PeerGroupAdvertisement 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
    StringWriter outWriter = new StringWriter();
    StructuredTextDocument docAdv =
        (StructuredTextDocument)adv.getDocument(new
                            MimeMediaType("text/xml"));
    try {
        docAdv.sendToWriter(outWriter);
    } catch (java.io.IOException e) {
```

```
            System.err.println("Can't Execute:
                        docAdv.sendToWriter(outWriter);");
        }
        System.out.println(outWriter.toString());

        // Let's end up with a line
        System.out.println(separator);
    }

    /** Starts the jxta platform */
    private PeerGroup startJxta() {
        PeerGroup myNetPeerGroup  = null;
        try {
            myNetPeerGroup=PeerGroupFactory.newNetPeerGroup();
        } catch ( PeerGroupException e) {
            // could not 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[]) {
        SecurePeerGroup satellaRoot = new SecurePeerGroup();
        System.exit(0);
    }
}
```

Chapter 8: JXTA Extension Package

The following chapter contains information on the JXTA-J2SE Extension Package.

The **ext:config** API is an optional JXTA extension that both eases and extends the core JXTA configuration principals. This is achieved through the combination of a comprehensive API implemented with optimal defaults encompassed with optional declarative models and UI constructs.

The principal **ext:config** constituents are:

- API
- Profile
- UI

Dependencies

At present, the required ext:config jars are the following:

- JXTA
  - jxta.jar
  - jxtaext.jar
- 3rd party
  - jdom.jar (pending removal, see #1453)
  - jaxen.jar (pending removal, see #1453)
  - jaxen-jdom.jar (pending removal, see #1453)
  - saxpath.jar (pending removal, see #1453)
  - swixml.jar (required only for UI)
  -

These jars can be obtained the code below and additionally from the **download** and **bootstrap process** referenced in the resources section.

Introduction

For the most part, JXTA applications are typically self configuring, taking as an exammple sample reference applications that act largely as "service consumers" by connecting to pre-existing JXTA networks. Saying that, some applications can benefit from fine tuning, augmenting and extending core JXTA configurations and this is the principal role of **ext:config**.

JXTA configuration information can be persisted in the form of a **PlatformConfig** object. Typically, for file based applications, a **PlatformConfig** file is resident in the relative **JXTA_HOME** directory before the JXTA Platform starts up. As such, JXTA configuration is a runtime precondition.

ext:config can be leveraged to construct, update, validate, extend, optimize and integrate JXTA configuration processes as best suited for the hosted application. **ext:config** is comprised of three principal domains:

- API
- Profile
- User Interface

The **ext:config/api** library is principally a traditional API with which one can construct, manipulate and obtain a valid JXTA configuration. The basic design of **ext:config/api** is largely that of "property sheet" as it provides a series of getter and setter methods, for the most part. The most important classes in **ext:config/api** are AbstractConfigurator[2] and Configurator.

The **ext:config/profile** is a declarative means to manage JXTA configuration processes by driving the afore mentioned **ext:config/api** via "profiles." The most important class in **ext:config/profile** is Profile.

The **ext:config/ui** provides a means to present user interface elements to manage JXTA configuration processes. **ext:config/ui** drives **ext:config/profile**.

API (aka ext:config)

The **ext:config API** is the core of the ext:config package with both the Profile and UI tiers layering above, in succession. A Configurator provides the ability instantiate and persist JXTA configuration state based upon declarative Profile and PlatformConfig information.

A Configurator serves primarily as a JXTA Configuration Bean, or property sheet, and implements very little attribute association logic beyond that of what is required to perform fundamental configuration integrity validation.

The principal constituents of a JXTA Configurator are:

- Peer Information
- Peer Transports
- Peer Services
- JXTA Network
- Configuration Extensions

Peer Information describes the local JXTA instance that a specified Configurator instance represents. The bulk of the Peer Information configuration data is optional and includes:

- name
- ID
- descriptor
- JXTA Home
- Log4J trace level
- security
- root certificate
- HTTP Proxy

Peer Transports describe the physical Address with which a Peer connects with the overall JXTA Network. Transports can be specified as incoming (aka server), outgoing (aka client) and both. Transport implementations include:

- TCP
- HTTP

Peer Services represent the JXTA Services that a specified Configurator instance will provision, which include:

- RendezVous
- Relay
- Proxy

The JXTA Network information specifies JXTA Network Services upon which a specified Configurator instance will rely upon, including:

- RendezVous
- Relays

The Configuration Extensions information includes extensible configuration features that includes end user provided Optimizer.

The principal API classes are:

- net.jxta.ext.config.AbstractConfigurator
- net.jxta.ext.config.Configurator

A number of supporting classes are also available that aid in the configuration management process.

The primary ext:config class a developer will use is AbstractConfigurator. This class implements the JXTA PlatformConfigurator interface and includes one abstract method:

```
public abstract PlatformConfig createPlatformConfig
  (PlatformConfigurator configurator)throws ConfiguratorException;
```

AbstractConfigurator implementations are notified via the **createPlatformConfig(PlatformConfigurator)** during JXTA Platform startup process when configuration data is required either as missing or incomplete. It is at this time applications can inject configuration details into the process.

As such, the following is a complete JXTA configurator minus the requisite application specify configuration details:

```
import net.jxta.ext.config.AbstractConfigurator;
import net.jxta.exception.ConfiguratorException;
import net.jxta.impl.peergroup.PlatformConfigurator;
import net.jxta.impl.protocol.PlatformConfig;

public class MyConfigurator
extends AbstractConfigurator {

   public PlatformConfig createPlatformConfig
      PlatformConfigurator pc) throws
      ConfiguratorException {

      // xxx: application specific configuration logic
      return pc.getPlatformConfig();
   }
}
```

In order to override the default JXTA Platform configuration process you register your AbstractConfigurator implementation with the Platform as the **configuration delegate**:

```
MyConfigurator.register(MyConfigurator.class);
```

In order to manipulate the configuration process during each and every startup one can override the **updatePlatformConfig(PlatformConfigurator)** method:

```
public PlatformConfig updatePlatformConfig(PlatformConfigurator configurator)
throws ConfiguratorException
```

The corresponding AbstractConfigurator implementation is thusly notified each time just prior to JXTA Platform start up.

A number of **resource management** APIs (from the class **AbstractConfigurator**) exist which serve to transfer named resources into the resulting **JXTA working directory**. Resources can be of any form and are accessed via unique names which, in turn, are mapped to file names hosted under the afore mentioned directory. This way, a developer can scope application data to distinct application instances.

Following is a resource management example derived from MyJXTA:

```
addResource("profile.xml", "/net/jxta/myjxta/resources/profile.xml");
addResource("log4j.xml", "/net/jxta/myjxta/resources/log4j.xml");
```

note: see #1449 (now supports private JXTA network)

```
addResource(PROFILE_KEY, "/net/jxta/myjxta/resources/profile.xml");
addResource("log4j.xml", "/net/jxta/myjxta/resources/log4j.xml");
```

With an **AbstractConfigurator** implementation in hand and registered with the JXTA Platform as the configuration delegate we now turn to the principal configuration class, aptly named **Configurator**.

The **Configurator** class serves to programatically create new and update existing **PlatformConfig** instances. As mentioned above, a **Configurator** can be managed via the **AbstractConfigurator** class or used standalone. For runtime environements, use of the **AbstractConfigurator** is encouraged as much of the file systems mechanics et al are managed by the **ext:config** framework. On the other hand, **Configurator** can be used to generate **PlatformConfig** objects for use elsewhere such as cluster configuration.

The lifecycle of a **Configurator** is typically quite short. Initially, a few key static constructs are invoked in order to establish a baseline **Configurator** instance. Firstly, a default **PlatformConfig** object is processed followed by the protected **Profile.SEED** profile. Lastly, the default **JXTA_HOME** is set to the **.jxta** directory hosted in the user's home directory, programmactically as follows:

```
setHome(new File(new File(System.getProperty("user.home")), ".jxta"));
```

note: Setting the **JXTA_HOME** via a static method has proven problematic and as such is currently under review. A likely refactor may include the **JXTA_HOME** as a constructor argument.

Upon instantiation, the **Configurator** object looks for a **PlatformConfig** in the specified **JXTA_HOME** and if found is processed to establish the relevant instance state. If a **PlatformConfig** is not found then a **Profile** by the name **profile.xml** hosted in the **JXTA_HOME** directory will be sought out and if found used to specify the relevant instance state. Failing to find either of the **PlatformConfig** and **profile.xml** files in the specified **JXTA_HOME** the default profile **Profile.DEFAULT** (which is effectively **Profile.EDGE**) will be used to establish the instance state.

Lastly, the provided constructor arguments are accessed accordingly. None of the **Configurator** constructors throw exceptions so creating a new instance is relatively ensured. The **principal** constructors are as follows:

1. Configurator(String name, String password)

2. Configurator(Profile)
3. Configurator(PlatformConfig)

note: Deprecated constructors are not included. In addition, a **JXTA_HOME** constructor may be added at a future time in order to address the static concern noted above.

The sole objective of the **Configurator** is to construct a viable **PlatformConfig** object. To this end there are two primary **PlatformConfig** generation methods:

1. public PlatformConfig getPlatformConfig() throws ConfiguratorException
2. public boolean save() throws ConfiguratorException
3. public boolean save(File pc) ConfiguratorException

note: The save() method simply invokes save(new File(getHome(), "PlatformConfig"))

The **ConfiguratorException** objects are chained such that **all** causes are retrievable via respective iterators.

At this point we are now armed with the ability to construct a trivial configurator:

```
import net.jxta.ext.config.Configurator;
import net.jxta.exception.ConfiguratorException;
import java.util.Iterator;
...
 Configurator c = new Configurator("usr", "pwd");
 try {
  c.save();
 } catch (ConfiguratorException ce) {
  for (Iterator c = ce.getCauses().iterator(); c.hasNext(); ) {
   Throwable t = (Throwable)c.next();
   System.out.println(t.getMesasge());
   t.printStackTrace(System.out);
  }
 }
```

There you have it. Effectively JXTA configuration accomplished in what amounts to 4 lines of code. Alas, true life is typically not so simple. as such we next turn to the number of **Configurator** getter/setter methods that provide for ultimate JXTA configuration tuning. Before doing so, let's dive down into the **PlatformConfig** processing internals a bit.

The **save** methods actually invoke the **getPlatformConfig()** method. As such, the only real function of the **save()** methods are to persist a **PlatformConfig**, as XML, to the relevant file system. With that, our attention turns back to the **getPlatformConfig()** method. In short, the following steps are executed in order resulting in either a valid **PlatformConfig** object or a causal **ConfiguratorException** exception:

1. normalization
2. optimization
3. validation
4. PlatformConfig construction

Taking each of these in turn we start with **normalization**. Normalization starts out by filling in any blank entries with the relvant defaults. No exceptions or faults are generated during this process. wip: describe address macro expansion

Next up, the **optimizer** kicks in by iterating the current and recently **normalized** *Configurator* state through all registered **Optimizers** in succession. wip: detail Optimizer interface, registration

Following the resultant **Configurator** state is exercised against a series of internal heuristics that check for **PlatformConfig** viability. This process, alone, can throw a **ConfiguratorException**.

Upon passing the **validity** stage all that remains is **PlatformConfig** creation whereby the **Configurator** state is transferred into a representative **PlatformConfig** instance.

At this point we should step back and inventory the principal getter/setter methods that are available prior to **PlatformConfig** generation. **Configurator** hosts a significantly large number of prototypical getter/setter methods that can be logically grouped as follows:

1. attributes (eg name, description, id, debug)
2. security (user, password)
3. services
    1. RendeZvous
    2. Relay
    3. Proxy
4. network
    1. TCP
    2. HTTP
    3. others as implemented
5. endpoint
6. miscellaneous

All of the above are invokable but typical use case calls for only focussing on the relevant domains after instantiating the **Configurator** with the relevant context (see Profile).

Profile (aka ext:config/profile)

Provides a means to declaratively manage Configurator processes by quantifing varying configuration classes into common domains in the form of "profiles."

A series of profile presets exist including:

name	description	Also see
EDGE	primarily a service consumer	EDGE_TCP, EDGE_HTTP
RENDEZVOUS	Rendezvous service	RENDEZVOUS_TCP, RENDEZVOUS_HTTP
RELAY	Relay service	RELAY_TCP, RELAY_HTTP
SUPER	primarily a service provisioner, (Rendezvous + Relay)	SUPER_TCP, SUPER_HTTP

LOCAL	primarily useful for development (standalone)	N/A
DEFAULT	equivalent to EDGE	

Most of the included profiles include subtle variations. Further, one can construct entirely new profiles that support specific application requirements.

All addresses are of the form URI. Addresses that do not specify scheme information will be defaulted accordingly to the respective context. Partial URI addresses will be templated with the respective context such as the local IP address, etc.

All fields have backing defaults enabling one to specify only the required overrides in order to construct complete configuration profiles.

The principal **ext:config/profile** class is:

- net.jxta.ext.config.Profile

For added flexibility, profiles can also be instantiated with either an URL or InputStream parameter the content of which conforms to the **profile schema definition** which can be found in both the Profile javadoc and **net.jxta.ext.config.resources.profile.xsd** file. This approach eases deployments by centrally managing application configuration meta data which is then accessed at application runtime.

Following is the prototypical **edge** profile which can serve as the basis for creating your specialized **Profile**:

```
<!DOCTYPE org.jxta:configuration>
<jxta>

  <peer descriptor="edge"/>
  <transport>
    <tcp>
      <address range="100">
        <multicast>udp://224.0.1.85:1234</multicast>
      </address>
      <publicAddress exclusive="false"/>
    </tcp>
    <http>
      <address range="100"/>
      <publicAddress exclusive="false"/>

      <proxy enabled="false"/>
    </http>
  </transport>
  <service>
    <relay>
      <outgoing enabled="true"/>
    </relay>
  </service>
</jxta>
```

Assuming the above profile was copied to a file, say **/tmp/myprofile.xml**, and modified as needed one could instantiate the resulting specialized **Profile** as follows:

```
import net.jxta.ext.config.Profile;
import net.jxta.ext.config.ResourceNotFoundException;
import java.net.URL;
import java.net.MalformedURLException;

...

  Profile p = null;
  try {
    p = new Profile(new URL("file:///tmp/myprofile.xml"));
  } catch (MalformedURLException mue) {
    mue.printStackTrace();
  } catch (ResourceNotFoundException rnfe) {
    rnfe.printStackTrace();
  }
```

The **edge** profile does not include all possible configuration options as some services are not enabled. Further, defaults, when used, are not duplicated thereby simplifying overall profile usage. Following is a comprehensive breakdown of the principal configuration elements:

(The following is a table that represents the profile.xml file.)

peer	peer description				
	descriptor	profile description			
	home	JXTA_HOME			
	trace	debug level			
	security				
	proxy				
network	subscribed services				
	rendezVous	subscribed rendezVous services			
		bootstrap	seeding URL		
		discovery	disallow usage of discovered rendezVous		
		address(es)	service URI		
	relays	subscribed relays services			
		bootstrap	seeding URL		
		discovery	disallow usage of discovered relays		
		address(es)	service URI		

transport	peer address(es)				
	tcp	TCP endpoint			
		address	endpoint URI		
		range	port range		
		multicast	muticast URI		
			enabled	multicast controller	
		publicAddress	public endpoint URI		
			enabled	public address controller	
		proxy	proxy address		
	http	HTTP endpoint			
		address	endpoint URI		
		range	port range		
		publicAddress	public endpoint URI		
		proxy	proxy address		
service	provisioned services				
	rendezVous	provisioned rendezVous service			
		enabled	rendezVous controller		
		autoStart	rendezVous provisioning policy in milliseconds		
			enabled	autoStart controller	
	relay	provisioned relay service			
		enabled	relay controller		
		queueSize	relay queue size		
		incoming			
			enabled	incoming controller	
			maximum	maximum messages	
			lease	lifetime in milliseconds	
		outgoing			

			enabled	outgoing controller	
			maximum	maximum messages	
			lease	lifetime in milliseconds	
	endpoint				
		queueSize	message queue size		
	proxy				
		enabled	proxy controller		
configuration					
	optimizer(s)	configuration optimization			
		class	optimizer class name		
			property(s)	optimizer attributes	
				name	value

Bringing it all together, one can trivially instantiate a **Configurator** with a selected **Profile** instance as a constructor argument as follows:

```
Configurator c = new Configurator(Profile.EDGE);
```

The resulting **Configurator** can readily be manipulated by any of the available APIs. As such, profiles are used to declaratively instantiate an application **Configurator** via a **configuration meta-model**. The resulting **Configurator** instance can, in turn, be adjusted via any of the provided APIs. Using **Profiles** one can readily experiment with varying models without having to recompile a line of code. Lastly, profiles are used internally by the **Configurator** class during instantiation to set the relevant defaults.

UI (aka ext:config/ui)

Lastly, the **ext:config/ui** tier provides an extensible UI above the afore mentioned **ext:config/profile** tier. It is done according to Java Swing's event model. A list of classes is provided below.

Class Summary	
AutoLoaderListener	Listener for rdv/rly autoloader events
ConfigTreeModel	Tree Model for ext:config:ui base tree
ConfiguratorListener	Listener for ext:config:ui events
HttpTransportTableModel	Table model for ext:config:ui http transport display
PeerTreeSelectionListener	ext:config:ui Tree selection event listener

ProfileComboModel	Combo box model for ext:config:ui JXTA profile selection
TcpAddressComboModel	Combo box model for ext:config:ui JXTA local address selection
TcpTransportTableModel	Table model for ext:config:ui tcp transport display
TraceComboModel	Combo box model for ext:config:ui JXTA trace level selection
TransportTableModel	Abstract base class for JXTA transport tables in ext:config:ui
UIConfigurator	Simple main application for ext:config:ui usage

Example Code

Following are the steps necessary to compile and run the included functional code samples. The prerequisites are:

1. J2SE (1.5 recommended0
2. Ant

See examples in the enclosed "configurator" directory or (obtain a working copy of the ext-config-lab.zip code samples from the web) and uncompress it accordingly followed by changing your working directory to that of the top-level **jxta** directory provided by the distribution. Running Ant with no arguments will display the usage message:

```
% ant
Buildfile: build.xml
usage:
     [echo] % ant [option]
     [echo]   configuration
     [echo]   profile
     [echo]   abstractconfigurator
     [echo]   connect
     [echo]   search
     [echo]   share
     [echo]   authentication
     [echo]   securesocket


     [echo]   all
     [echo]   clean
```

The excercises are contained under the **src** directory and organized as incremental and progressive code samples. The **lib** directory includes the necessary JXTA and ext:config libraries:

JXTA jars:

1. bcprov-jdk14.jar
2. javax.servlet.jar
3. jxta.jar
4. log4j.jar
5. org.mortbay.jetty.jar

ext:config jars:

1. jaxen-core.jar
2. jaxen-jdom.jar
3. jdom.jar
4. jxtaext.jar
5. saxpath.jar

note: the ext:config jars will likely soon be replaced with the JAXP 1.3 equivalents
To run an excercise simply provide the excercise directory as an Ant argument:
% ant configuration profile

```
Buildfile: build.xml
http.proxy:
socks.proxy:
prepare:
configuration:
configuration.compile:

compile:
    [mkdir] Created dir: C:\jxta_cvs\guide\src\guide_v2.3\april_7_2005\jxta_devg
uide\Configurator\build\configuration\classes
    [javac] Compiling 1 source file to C:\jxta_cvs\guide\src\guide_v2.3\april_7_
2005\jxta_devguide\Configurator\build\configuration\classes
    [javac] Note: C:\jxta_cvs\guide\src\guide_v2.3\april_7_2005\jxta_devguide\Co
nfigurator\src\configuration\SimpleConfigurator.java uses or overrides a depreca
ted API.
    [javac] Note: Recompile with -Xlint:deprecation for details.

configuration.run:
run:
     [echo] home: C:\jxta_cvs\guide\src\guide_v2.3\april_7_2005\jxta_devguide\Co
nfigurator/build/configuration
     [java] create PlatformConfig with the following properties:
     [java]   name = MyPeerName
     [java]   description = MyPeerDescription
     [java]   principal = MyPeerPrincipal
     [java]   password = MyPeerPassWord
     [java] saving PlatformConfig to: file:/C:/jxta_cvs/guide/src/guide_v2.3/apr
il_7_2005/jxta_devguide/Configurator/build/configuration/home/

http.proxy:
socks.proxy:
prepare:
profile:
```

```
profile.compile:
compile:

profile.run:
run:
    [echo] home: C:\jxta_cvs\guide\src\guide_v2.3\april_7_2005\jxta_devguide\Co
nfigurator/build/profile
    [java] create PlatformConfig with the following properties:
    [java]   profile = profile.xml
    [java] saving PlatformConfig to: file:/C:/jxta_cvs/guide/src/guide_v2.3/apr
il_7_2005/jxta_devguide/Configurator/build/profile/home

BUILD SUCCESSFUL
Total time: 12 seconds
```

Caution editing your profile.xml

You might encounter the following exception. This can be caused by a certificate tag in the peer section. In this case, the problem was that the that was there, but no cert address was provided. The actual error thrown deep in the java security was that the key was too long("Could not parse certificate"). In fact there was no key, just gibberish. Removal of the tag solves the problem.

```
<ERROR 2005-07-22 13:29:33,526 JxtaConfig::File;)V:187> Error initializing configuration
java.lang.IllegalArgumentException: Failed to process cert
        at net.jxta.impl.protocol.PSEConfigAdv.setCert(Ljava.lang.String;)V(PSEConfigAdv.java:355)
        at net.jxta.ext.config.Configurator.setRootCertificateBase64(Ljava.lang.String;)V(Configurator.java:1271)
        at net.jxta.ext.config.Configurator.process(Lnet.jxta.ext.config.Profile;Z)V(Configurator.java:2416)
        at net.jxta.ext.config.Configurator.process(Lnet.jxta.ext.config.Profile;)V(Configurator.java:2345)
        at net.jxta.ext.config.Configurator.<init>(Ljava.net.URI;Lnet.jxta.ext.config.Profile;)V(Configurator.java:431)
        at com.cluck.jxta.util.JxtaConfig.<init>
(Ljava.lang.String;Ljava.lang.String;Ljava.lang.String;Ljava.lang.String;Ljava.io.File;)V(JxtaConfig.java:84)
      ....
```

Resources

1. ext:config API - note: package net.jxta.ext.config
2. ext-config.zip
3. JXTA Downloads
4. JXTA Bootstrap
5. JXTA jar dependencies
6. Daniel Brookshier's Autoconfig example

Chapter 9: Miscellaneous

1. Configuration

A) How to run multiple JXTA instances on the same machine

There are two ways to run multiple copies of JXTA simultaneous on the same machine.

Way #1, setting system property

% java -DJXTA_HOME = “alternative directory”

or

% ant -Dnet.jxta.jxta_home=peer2 run

A directory named "peer2" will then be created that hosts the second peer's runtime information. You will likely have to ensure that unused ports are registered so as to ensure proper execution. Some applications, like MyJXTA2, perform preliminary configuration processes to ease this process although it is not entirely fool proof.

Way #2, using ant scripts.

The basic idea here is to launch several peers with different pre-configured names at the push of a button:

```
<target name="launch swarm" description="runs lots of peers all at once'">
  <java classname="YourAppNameHere" failonerror="yes" fork="yes">
    <jvmarg value="-DJXTA_HOME=peer_name_0"/>
    <classpath refid="classpath" />
  </java>
  <java classname="YourAppNameHere" failonerror="yes" fork="yes">
    <jvmarg value="-DJXTA_HOME=peer_name_1"/>
    <classpath refid="classpath" />
  </java>
  <java classname="YourAppNameHere" failonerror="yes" fork="yes">
    <jvmarg value="-DJXTA_HOME=peer_name_2"/>
    <classpath refid="classpath" />
  </java>
  <java classname="YourAppNameHere" failonerror="yes" fork="yes">
    <jvmarg value="-DJXTA_HOME=peer_name_0"/>
    <classpath refid="classpath" />
  </java>
</target>
```

B) Running two or more peers on a stand alone machine without a network connection.

If you want to use peers with TCP on a single host you should:

Disable the HTTP transport.

Select the manual ip address for TCP and enter 127.0.0.1.

Use different ports for each peer.

Now part of the cause of my trouble is that I use dhcp for all my connections (powerbook with OS X). So when i'm not on a network i don't have an ip address other than localhost. If you have a machine with a fixed ip this probably wouldn't be a problem.

C) JXTA and HTTP Proxies

This topic relates to configuring JXTA to communicate via HTTP proxy servers and configuration for those proxies that may require Authentication:

- How do I know if I need to enable proxy support?
- How do I configure proxy support?
- How do you set the proxy authentication user ID and password programatically?
- How do you set the proxy authentication user ID and password from the java or Javaw executable?
- How do you set the proxy authentication user ID and password with Ant?
- What are the Proxy Authenitcation user ID and password used for?

JXTA and HTTP Proxies

In order enable JXTA peers to communicate across firewalls JXTA supports the use of HTTP proxies, the same proxies used by web browsers. Some proxy server configurations may also require authentication.

Do I need a Proxy?

If you need to use an HTTP proxy with your web browser then you will need to use a proxy with JXTA as well. Most web browers 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.

If your web browser is using an HTTP proxy then you should write down the HTTP proxy address and port number. You will need to provide this information to JXTA.

Proxy Support Configuration

Configuring a JXTA peer to use an HTTP proxy is straight forward.

1. Enable the "HTTP Message Transport" on the "Advanced" panel of the JXTA configuration dialog. * If all of the other JXTA peers are located outside of your firewall the you may disable incoming connections. The firewall prevents other peers from reaching you. * You will need to enable use of a JXTA relay. This is specified on the "Rendezvous/Relay" options page of the JXTA Configuration Dialog.
2. Tell Java the proxy address and port number. There a couple of ways to do this detailed below.

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

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

Setting Proxy and Proxy Authentication Programatically

You can also set the proxy configuration and authentication Java System Properties within a Java application. The following method can be used to set these. This method must be called before JXTA is initialized. Including a password in code, as shown below, is a really really bad idea. Your application should instead prompt for the password

```
/**
 *    set the proxy and proxy authentication
 */


public void enactProxyAuthentication() {
    System.setProperty("http.proxyHost", "proxy.mycompany.com");
    System.setProperty("http.proxyPort", "8080");
    System.setProperty( "jxta.proxy.user","usr");
    System.setProperty( "jxta.proxy.password","pwd");
}
```

Some Java configurations, such as Applets, JSPs, Java WebStart (JNLP) or the use of security managers may disallow the setting of System Properties. In these cases you can usually specify System Properties as part of the configuration of the execution environment.

Setting Proxy and Proxy Authentication with Ant

To set the Java systemp properties in Ant, use the **sysproperty**. Set the system properties: `http.proxyHost`, `http.proxyPort`, `jxta.proxy.user`, and `jxta.proxy.password`

```
<target depends="compile" description="Run the Swing app" name="run_RDV">
  <property name="testing.dir" value="../test/test3"/>
  <mkdir dir="../test"/>
  <mkdir dir="${testing.dir}"/>
  <mkdir dir="${testing.dir}/.jxta"/>
  <touch file="${testing.dir}\.jxta\reconf"/>
  <java classname="com.cluck.jxta.util.SmartRDV" classpathref="jxta.run.classpath" dir="${testing.dir}"
failonerror="true" fork="true">
    <sysproperty key="http.proxyHost" value="proxy.mycompany.com"/>      <-- set proxy -->
    <sysproperty key="http.proxyPort" value="8080"/>
    <sysproperty key="jxta.proxy.user" value="usr"/>          <-- set proxy user -->
    <sysproperty key="jxta.proxy.password" value="pwd"/>  <-- set proxy password -->
      <arg value="rdv"/>
      <sysproperty key="JxtaID" value="DeadBeefDeafBabaFeedBabe000000010306"/>
    </java>
</target>
```

D) Logging configuration with Log

Dealing with log4j under JXTA can be a bit challenging, mainly because the platform logs out of sight of your application. I set a goal of figuring out how the platform is initialized with respect to log4j, then modify that initial configuration to suit my needs such that logging, layouts, and appenders exercised at runtime would be a bit more transparent. No Chainsaw considerations are taken up here - only log4j considerations. This is a very basic treatment that hopefully will get us started.

The modifications of interest: a) suppress all console logging output, b) configure a rolling file appender for the platform to use, and c) create loggers for the app, and which uses the same rolling appender the platform uses.

Create a Log4j properties template

Using `platform/binding/java/log4j.properties` as a starting point, we have

```
#
log4j.rootLogger=FATAL, A1
# JXTA platform specific
log4j.appender.A1=org.apache.log4j.RollingFileAppender
log4j.appender.A1.File=${user.home}/myapp/applog.txt
log4j.appender.A1.MaxBackupIndex=1
log4j.appender.A1.MaxFileSize=1000KB
log4j.appender.A1.Threshold=DEBUG
log4j.appender.A1.layout=org.apache.log4j.PatternLayout
log4j.appender.A1.layout.ConversionPattern=<%-5p %d{ISO8601} %C{1}::%M:%L> tg=<%t> %m>%n
log4j.logger.net.jxta.impl.peergroup.ConfigDialog=INFO
log4j.logger.net.jxta.impl.peergroup.DefaultConfigurator=INFO
log4j.logger.net.jxta.impl.peergroup.NullConfigurator=INFO
# App specific
log4j.logger.org.yourdomain.yourpackage=INFO
```

Notice how there is no console appender - only a file appender for all loggers. File logging goes to `$HOME/myapp/applog.txt`. This includes logging in hierarchy org.yourdomian.yourpackage, as the app logger will inherit the root logger's appender details.

Include Log4j properties template in app jar file

Place this properties file in `org/yourdomain/yourpackage/log4j.properties` and include it in the application jar. You can verify that the property file is in the app jar file by doing something like

```
$ jar tf myapp.jar |grep log4j.properties
org/yourdomain/yourpackage/log4j.properties
```

Configure runtime instance of Log4j

In your app **before** you start the platform with `PeerGroupFactory.newNetPeerGroup()`, configure Log4j itself

```
private void initLogger() {
```

```
  System.setProperty("log4j.defaultInitOverride", "true");
  ClassLoader cl = this.getClass().getClassLoader();
  InputStream is = cl.getResourceAsStream("org/yourdomain/yourpackage/log4j.properties");
  try {
    Properties p = new Properties();
    p.load(is);
    String ll = rtProps.getProperty("loglevel", "info").toLowerCase();
    if( ! (ll.equals("info") || ll.equals("warn") || ll.equals("debug") ) )  // optional
      ll = "info";
    p.setProperty("log4j.logger.org.yourdomain.yourpackage", ll);
    p.setProperty("log4j.logger.net.jxta.impl.peergroup.ConfigDialog", ll);
    p.setProperty("log4j.logger.net.jxta.impl.peergroup.DefaultConfigurator", ll);
    p.setProperty("log4j.logger.net.jxta.impl.peergroup.NullConfigurator", ll);
    PropertyConfigurator.configure(p);
  }
  catch (Exception e) {
    // fallback to whatever is in the resource
    System.setProperty("log4j.configuration", "org/yourdomain/yourpackage/log4j.properties");
  }
  logger = Logger.getLogger( this.getClass() );
}
```

where we've assumed Properties `rtProps` contains a key named `loglevel` that has been persisted by the app to hold the app and platform loglevel between runtime invocations. In this model, therefore, `org/yourdomain/yourpackage/log4j.properties` is a template only.

Notice how we suppress the default initialization of log4j with

System.setProperty("log4j.defaultInitOverride", "true");

This model therefore takes explicit control of the log4j init with

PropertyConfigurator.configure(p);

See Log4jDocs for more info on log4j initialization.

Tell ExtConfig

Assuming we are using ExtConfig, call

```
configurator.setTrace(Trace.INFO);
```

for log level "info", etc. I'm not certain of the nature of the interplay between the log levels for the platform classes, e.g, `net.jxta.impl.peergroup.ConfigDialog`, and the <Dbg> tag in the `PlatformConfig`. In other words, must we even explicitly set the log level for these classes in `initLogger()`, or can we let the <Dbg> value handle that? Dunno. I haven't tried it. There may be a more elegant way to configurare logging for the platform and the app that what I've shown here. But this model works for me. I like hiding the log properties in a jar file, where they are more difficult to accidentally modify. If nothing else, the technique shows how to suppress console log output and redirect to a file.

Start the app

Finally, start the platform and the app with it with

```
PeerGroupFactory.newNetPeerGroup();
```

E) Rendezvous & Relay configuration

Running a public rendezvous/relay - System and network requirements

---

Minimum System requirements :

- 1Ghz processor
- 384MB ram
- Solaris x86, Sparc, JDS, or Windows 2000
- Java 1.4.2 or higher

Minimum Network requirements :

- 768Kb Up, 384Kb Down
- Directly addressable on the Internet
- Must allow incomming connections for tcp, http or both if such transports are configured

Minimum Runtime requirements :

- iViewRendezvous application see JXTAnetmap Project
- Set file limit to 10240
- Set minimum, maximum vm heap while taking into consideration system memory requirements to avoid memory thrashing
- Join the JXTA Public Rendezvous Project and subscribe to `discuss@public-rendezvous.jxta.org` mailing list.
- Send an email to discuss@public-rendezvous.jxta.org with the node public address, and ports

OS	Mem	Min	Max
Solaris	768MB	64m	384MB
JDS	768MB	64m	384MB
Win2K	768MB	64m	384MB

Configuration notes :

- Setting fd limits on Linux

  /etc/security/limits.conf

```
*       soft    nofile  10240
*       hard    nofile  65535
```

  Verify the settings after a system reboot using the "ulimit -a" command

When setting up a peer as a JXTA Rendezvous or Relay it is important to make sure that the host machine is correctly configured for best performance.

Important Considerations

1. Make sure you have the current OS patches for the JDK you are using.
2. Ensure you have the latest patch level for the JDK you are using.
3. Set Process Limits on Your Server
    1. File Limit (example on Unix or Mac OSX: ulimit -n 32768)
    2. Socket Connection Limit
    3. Process Memory Size Limit
4. Set Java VM Parameters
    1. Set `-server` mode.
    2. Set `-Xms` to at least 64MB. (note that on some systems this is ignored at a ceertain point and must be extended at the system or kernel)
    3. Set `-Xmx` to a value appopriate amount based on how much memory you have.
5. Set JXTA Parameters
    1. Except for very rare network bridging configurations, no peer should **ever** be configured to be both a Relay client and a Relay server.
    2. Except for very rare network bridging configurations, no peer should **ever** be configured to be both an HTTP client and an HTTP server.
    3. TCP/IP Multicast should be disabled for all Rendezvous and Relay servers, if more than one Rendezvous/Relay exists on the same sub-net
    4. "Use Only Seeds" should **not** be enabled for Rendezvous and Relay servers.
    5. In situations where a node is heavily utilized for it's relay services, it's highly recommended that such node should only offer relay services (no rendezvous).

Sub PeerGroup RDV Considerations for Edge Peers

1. General Issues with sub PeerGroup RDV
    1. Current non-server licesnsed MS Windows operating system have limits to incomming (10?) and outgoing connections. This means that direct connection are limited and may cause most connectivity to be routed via a Relay. The implication also means that a peer running as a RDV in a sub peer group cannot have a large number of direct connections but instead is connected to clients via the Relay, including other RDV in the sub peer group.
    2. File Limit mentioned above becomes a hard limit to the number of peers that can be hosted by a sub peer group
    3. To efficiently manage a sub peer group RDV you should reduce the number of RDV clients to a managable limit of the specific operating system and configuration. This will cause more RDV to be created or a RDV connected that has excess capacity.
    4. Populating a network with high capacity Edges for each sub group can reduce latency. In other words avoid RDVs running on small machines with poor bandwidth.

F) RDV AccessList

Relay and Rendezvous access control :

Relay and Rendezvous access lists were introduced as part of JXTA platform release 2.3.2. This feature provides the ability to dynamically (at runtime) control the list of rendezvous and relay peers may offer such services in the infrastructure group, the mechanism controls span on the wire service messages, and locally cached resources.

When enabled on a Rendezvous/Relay peer, only those with a "grant" level access will be allowed to join the PeerView[?], Relay cache, and their resource advertisement cached. On an edge peer, only those with a grant access will be accepted as service providers. A typical deployment would include access lists on rendezvous/relay peers, and no ACL on edges, a completely restricted deployment would include access list deployed across all nodes.

- Access Control is agnostic to a peer's role (rendezvous, or edge) if no lists exist ($JXTA_HOME/relayACL.xml, rendezvousACL.xml) "grant all" is assumed
- Denial of access must be explicit i.e. a peer entry must define the attribute "access=deny"
    - lack of the attribute denotes "access=grant"
- A Global <grantAll>true</grantAll> overrides any deny permissions
- Dynamic Access Control refresh (no restart required), access lists may updated on a live system
- Rendezvous, and Relay resource access control
    - No relay, or rendezvous peers are allowed to join the view through the PeerView--Relay protocols without access permissions

Installation :

- Edit Ready made templates provided by the platform with the desired list of peer ID's
    - platform/binding/java/contrib/rendezvousACL.xml and platform/binding/java/contrib/relayACL.xml

Sample Access Lists :

rendezvousACL.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:XACL>
<jxta:XACL xmlns:jxta="http://jxta.org">
    <description>Rendezvous Access List</description>
    <peer name="JXTA.ORG 36" access="grant">
        urn:jxta:uuid-59616261646162614A7874615032503350F16FEF24D0452A905EB6EEB108FE0903
    </peer>
    <peer name="JXTA.ORG 37" access="grant">
        urn:jxta:uuid-59616261646162614A787461503250337013A5BB82D24D2B9463AE7220B1582E03
    </peer>
    <peer name="JXTA.ORG 38" access="grant">
        urn:jxta:uuid-59616261646162614A78746150325033D15CB418EF884A9A8B01A7C6B770E89403
    </peer>
    <peer name="JXTA.ORG 39" access="grant">
        urn:jxta:uuid-59616261646162614A7874615032503635BD5A2D4A9483B87AA885EC92E3D4C03
    </peer>
</jxta:XACL>
```

relayACL.xml:

```
<?xml version="1.0" encoding="UTF-8"?>
<!DOCTYPE jxta:XACL>
```

```
<jxta:XACL xmlns:jxta="http://jxta.org" >
    <description>Relay Access List</description>
    <grantAll>false<?/grantAll>
    <peer name="JXTA.ORG 36">
        urn:jxta:uuid-59616261646162614A7874615032503350F16FEF24D0452A905EB6EEB108FE0903
    </peer>
    <peer name="JXTA.ORG 37" access="grant">
        urn:jxta:uuid-59616261646162614A787461503250337013A5BB82D24D2B9463AE7220B1582E03
    </peer>
    <peer name="JXTA.ORG 38" access="grant">
        urn:jxta:uuid-59616261646162614A78746150325033D15CB418EF884A9A8B01A7C6B770E89403
    </peer>
    <?peer name="JXTA.ORG 39" access="grant">
        urn:jxta:uuid-59616261646162614A7874615032503635BD5A2D4A9483B87AA885EC92E3D4C03
    </peer>
</jxta:XACL>
```

G) Application Testing

TestingConfigurator.java:

```
import net.jxta.ext.config.TrivialConfigurator;
public class TestingConfigurator extends TrivialConfigurator {
   public TestingConfigurator() {
      // use JXTA_HOME as the peer name:
      this.setName(TestingConfigurator.getHome().getName());
   }
}
```

Put TestingConfigurator.java into your project and add this next line right before you start the platform:

```
PeerGroupFactory.setConfiguratorClass(TestingConfigurator.class);
```

H) Private JXTA Networks

This topic is related to either isolating your JXTA network for privacy or for testing without affecting the public network. Testing is probably more important because you should "NEVER" add a peer to the public network that acts as a RDV or acts as Relay (this will not even be possible in some networks, including the public network, in the future versions of JXTA -post 2.2.1). The reason is that only managed peers should be setup to act this way. Testing also causes problems if these peers are brought up and down.

If we are testing, we also might want to be sure that we are in control if the network and have the best response. By having out own RDV we can be sure all the tests work and are not affected by the public network. By creating the public network we ensure there are no public peers attaching and using our RDV and Relay or that our test peers are not talking to the public RDV/Relay that may be busy or even using a different version of JXTA.

A private network can also be used for a corporation. This allows the corporation to allocate RDV/Relays and isolate peers that use them. This does two things that are important to many companies: First is isolation and thus a level of security. Second, this allows you to monitor and manage the RDV and Relay peers to ensure a level of service for your clients.

The best solution is to place a config.properties file in your .jxta directory. Do this with all your peers and the RDV peer. This prevents all the peers from contacting the public peer groups even if the seeds are downloaded. Note that only one peer needs to download the seeds to infect the group and add it to the public network! That's why this is the better solution because it is true isolation.

Note that this explicitly disables the capacity of a peer to reach the public NetPeerGroup[?].

Here is what should be in config.properties (as of 2004/08/13):

```
NetPeerGroupID=uuid-????????????????????????????????
NetPeerGroupName=MyNameForNetGroup
NetPeerGroupDesc=My desc for Infrastructure Group
```

NOTE: the builds PRIOR to 2004/08/13 must not include "jxta" namespace in front of NetPeerGroupID[?]. The formating of this property should look like this:

```
NetPeerGroupID=jxta:uuid-????????????????????????????????
```

How do you generate the peer group ID? Below is how I do it so that I can generate it on the fly. You can either do this before you start jxta and write the file or print the ID and paste it into the config.properties file. Note that if it is printed, it is prefixed with "urn:" which must be removed. For builds after 2004/08/13 "urn:jxta:" must be removed.

```
 public static final net.jxta.peergroup.PeerGroupID createInfrastructurePeerGroupID(String clearTextID, String function){
    LOG.info("Creating peer group ID =  clearText:'"+clearTextID+"' , function:'"+function+"'");
    byte[] digest = generateHash(clearTextID, function);
    net.jxta.peergroup.PeerGroupID peerGroupID = IDFactory.newPeerGroupID(  digest );
    return peerGroupID;
 }
 /**
  * Generates an SHA-1 digest hash of the string: clearTextID+"-"+function or: clearTextID if function was blank.<p>
  *
  * Note that the SHA-1 used only creates a 20 byte hash.<p>
```

```
     *
     * @param clearTextID A string that is to be hashed. This can be any string used for hashing or hiding data.
     * @param function A function related to the clearTextID string. This is used to create a hash associated with
clearTextID so that it is a uique code.
     *
     * @return array of bytes containing the hash of the string: clearTextID+"-"+function or clearTextID if function was
blank. Can return null if SHA-1 does not exist on platform.
     */
    public static final byte[] generateHash(String clearTextID, String function) {
        String id;

        if (function == null) {
            id = clearTextID;
        } else {
            id = clearTextID + functionSeperator + function;
        }
        byte[] buffer = id.getBytes();

        MessageDigest algorithm = null;

        try {
            algorithm = MessageDigest.getInstance("MD5");
        } catch (Exception e) {
            LOG.error("Cannot load selected Digest Hash implementation",e);
            return null;
        }


        // Generate the digest.
        algorithm.reset();
        algorithm.update(buffer);

        try{
            byte[] digest1 = algorithm.digest();
            return digest1;
        }catch(Exception de){
            LOG.error("Failed to creat a digest.",de);
            return null;
        }
    }
```

Setup for using JXTA Netmap

1. From the JXTA NetMap Project page, download *NetMapRendezvous.jar* and *netmap.jar*
2. Create 2 directories: one for the rendezvous, and the other for the graphical visualizer and move the corresponding jars into these directories
3. In both directories, create a .jxta directory and copy the config.properties file from above.
4. From the rendezvous directory, run: *java -jar NetMapRendezvous.jar.* This will launch the configuration screen.
    1. Under the Advanced section, configure the TCP and HTTP Transport sections. Note that the default ports on both of these is 9900. These ports CANNOT be the same value if you plan on enabling both transports. Also, if you populate the optional public address, make sure the port numbers match up to the values you picked.
    2. User the Rendezvous/Relay section, remove the default rendezvous seeding URI and either provide your own URI here or manuall add your rendezvous as a seed peer.
    3. Click OK
5. From the viewer directory, type: *java -jar Desktop/netmap.jar*

I) JNLP

Java WebStart is amazing. It is just as simple as that. Installation is trivial and the opportunities to endless. I just combed the JNLP Specification and am impressed with the improvements contained within since I had last tooled around with JNLP.

Enough WebStart gushing ... how am I using JNLP in my day-to-day.

Here's the skinny. I have a rich client that sits on top of JXTA. We've been deploying as a binary, source and cvs for quite some time now and even JNLP. Upon updating to J2SE 1.4.2 I took the time, as so should you, getting familiar with the included WebStart mods. Firstly, adding WebStart to your (*nix) env (as if there is any other) is as trivial as adding {j2se}/jre/javaws to your PATH. Next up, I simply had to dust off my original JNLP file.

For my particular app I distribute a single jar, that is myjxta-2.2a.jar, yet I largely (and happily) depend upon JXTA and all it's constituents with which it is comprised, some of which may (or may not) be signed 3rd party jars in turn and can vary over time, etc.

JNLP 1.0.1 (and possibly earlier) inlcludes a really cool "extension resource" such that any one app can reference zero or more "component" JNLP distributions which can, in turn, further reference zero or more component JNLP distributions. The result, of which, is that I, as a JXTA application developer, need no longer care about JXTA deployment details, upgrades, jar signing, etc. To be honest, all of the above are issues that one, to an extent, should be aware of but my point is that with this very enabling deployment model I need not care as much about the constituent details as I was forced to without JNLP, and to that end I "throw mad props" to WebStart/JNLP team.

As a result, my new JNLP file is elegantly simple and included here:

```
<?xml version="1.0" encoding="utf-8"?>
<jnlp spec="1.0+"
    codebase="http://download.jxta.org/myjxta2/2.2a"
    href="myjxta.jnlp">
  <information>
    <title>MyJXTA</title>
    <vendor>myjxta2.jxta.org</vendor>
    <homepage href="http://myjxta2.jxta.org"/>
    <description>MyJXTA: chat, share ... change the world</description>
    <icon href="news.gif" width="10" height="16"/>
    <offline-allowed/>
  </information>
  <security>
    <all-permissions/>
  </security>
  <resources>
    <j2se version="1.4+"/>
    <jar href="lib/myjxta-2.2a.jar" main="true" download="eager"/>
    <extension href="http://download.jxta.org/platform/2.2/platform.jnlp"/>
    <extension href="http://download.jxta.org/cms/2.2/cms.jnlp"/>
  </resources>
```

```
    <application-desc main-class="net.jxta.myjxta2.Main"/>
</jnlp>
```

Lastly, here's the afore mentioned JNLP file in action : MyJXTA 2.2a

I constructed the underlying Platform, Security and CMS JNLP files as well and while they are a bit more involved they are logicaly scoped, can vary independently and are readily re-usable ... simply by adding the relevant "extension resources" as shown above.

2. Application Development

A) Lightweight PeerGroup

Introduction

The JXTA J2SE platform provides a very rich implementation of PeerGroup which allows support for multi platform. In addition, the JXTA PeerGroup automatically instantiates the default standard JXTA protocols such as Discovery, Pipe, RendezVous, etc. Finally, the JXTA PeerGroup implementation are shareable, using a PeerGroup factory and registry. While all those features are very useful, there is a need for a lighter PeerGroup implementation for JXTA applications that do not require those features:

- the standard JXTA PeerGroup implementation requires a larger memory footprint, threads and initialization latency as each peergroup runs its own instance of the peergroup services;
- The standard JXTA PeerGroup implementation requires the use of low-level Service API in order to configure peergroups for specific needs (such as adding or removing peergroup services). The PeerGroup API only provides PeerGroup cloning as convenience method;

The goal of the JXTA Light Weight PeerGroup is to provide a convenience and friendlier PeerGroup API that addresses limitations of the current API while adding capability such as allowing children peergroups to re-use parent peergroup services.

- the Light Weight PeerGroup does not provide hooks for multi platform automatic support: code binding is to the responsibility of the application;
- the Light Weight PeerGroup allows application to easely set, change, add or delete PeerGroup services as well as providing implementation;
- the Light Weight PeerGroup is extendable; the Light Weight PeerGroup can be directly instantiated by application (no factory).

Design

The Light Weight PeerGroup is implemented with a base class, LightWeightPeerGroup . This class implements the PeerGroup interface as follow:

- use a provided PeerGroup as a parent PeerGroup;
- by default, all services of a LightWeightPeerGroup are those of the parent PeerGroup;
- provides (null) stubs implementation for all the standard JXTA PeerGroup module supports: LightPeerPeerGroups can be used just like standard JXTA PeerGroups;
- provides a constructor (no factory).

While the LightWeightPeerGroup class is intended to be extended, it can be directly used by applications that desire to encapsulate an existing PeerGroup (the parent PeerGroup). The resulting LightWeightPeerGroup is then a weak reference to the parent PeerGroup with a new PeerGroup identifier. The following example shows such a scenario:

```
import net.jxta.peergroup.LightWeightPeerGroup;

/**

 * Creates a new PeerGroup object, cloning the parent PeerGroup and

 * describing the new PeerGroup with a provided PeerGroupAdvertisement.

 * @param parent parent PeerGroup

 * @param newadv PeerGroupAdvertisement of the new PeerGroup
```

```
 **/

PeerGroup dupPeerGroup (PeerGroup parent, PeerGroupAdvertisement newadv) {

    // Creates a new LightWeightPeerGroup with the given advertisement

    PeerGroup group = new LightWeightPeerGroup (newadv);

    // Initialize the PeerGroup.

    group.init (parent, null, null);

    return group;

}
```

Tutorial 1: Discovery Access Right PeerGroup

The purpose of this tutorial is to show the usage of LightWeightPeerGroup as an overload of an existing purpose. The following class implements a PeerGroup cloning an existing PeerGroup, adding access right control to the Discovery Service.

```
1  import net.jxta.peergroup.PeerGroup;
2  import net.jxta.peergroup.LightWeightPeerGroup;

3  import net.jxta.discovery.DiscoveryService;

4  public class ReadOnlyPeerGroup extends LightWeightPeerGroup {

5    private PeerGroup group = null;

6    private ReadOnlyDiscoveryService discovery = null;

7    public ReadOnlyPeerGroup (PeerGroup group) {

8        super (group.getPeerGroupAdvertisement());

9        this.group = group;

10       super.init (group, null, null);

11       this.discovery = new ReadOnlyDiscoveryService (group);

12   }

13   public DiscoveryService getDiscoveryService () {

14       return discovery;

15   }
```

```
16   public Service lookupService(ID name) {

17     throw new RuntimeException ("Not allowed with this PeerGroup");

18   }

19 }
```

Cloning a PeerGroup

The constructor of the class *ReadOnlyPeerGroup* takes only one argument, the parent PeerGroup. Line 8 invokes the LightWeightPeerGroup constructor, giving the parent PeerGroup, and the parent PeerGroup's advertisement: the cloned PeerGroup represented by the instance of ReadOnlyPeerGroup will have the same advertisement as the parent.

Overloading a PeerGroup service

Line 10, the constructor initializes the LightWeightPeerGroup. Note that since the LightWeightPeerGroup has already associated a PeerGroupAdvertisement with the PeerGroup, and since LightWeightPeerGroup does not provide support for multiple implementation, no assigned identifier or implementation advertisement is provide.

Line 11, the constructor instantiates a ReadOnlyDiscoveryService which is a DiscoveryService stub forbidding publishing.

Line 13, the method getDiscoveryService() is overloaded to return the ReadOnlyDiscoveryService instead of the parent's group Discovery Service.

Line 16, the *lookupService* method is overloaded and disabled, forbidding the application to access to the Discovery Service other than using *getDiscoveryService* method. Of course, an alternative implement ion could filter out lookup for the Discovery Service.

Overloading a PeerGroup Service

The following does not purely relate to LightWeightPeerGroup but is provided for completeness of the tutorial. The class *ReadOnlyDiscoveryService* is a weak reference to the provided Discovery Service, throwing exception for an publish attempt.

```
import java.util.Enumeration;

import java.io.IOException;

import net.jxta.discovery.DiscoveryListener;

import net.jxta.discovery.DiscoveryService;

import net.jxta.document.Advertisement;

import net.jxta.id.ID;

import net.jxta.peergroup.PeerGroup;

import net.jxta.service.Service;
```

```
public class ReadOnlyDiscoveryService implements DiscoveryService {

    private DiscoveryService disco;

    public Service getInterface() {

        return this;

    }

    public Advertisement getImplAdvertisement() {

        return disco.getImplAdvertisement();

    }

    public ReadOnlyDiscoveryService(PeerGroup group) {

        disco = group.getDiscoveryService();

    }

    public int getRemoteAdvertisements(String peer, int type,

                                       String attribute, String value, int
threshold) {

        return disco.getRemoteAdvertisements(peer, type, attribute,

                                            value, threshold);

    }

    public Enumeration getLocalAdvertisements(int type, String attribute,

            String value) throws IOException {

        return disco.getLocalAdvertisements(type, attribute, value);

    }

    ...

    public void publish(Advertisement adv,

                        int type,

                        long lifetime,

                        long lifetimeForOthers)
```

```
    throws IOException {

        throw new IOException ("publish is disabled - Read only PeerGroup");

    }

    public void remotePublish(Advertisement adv, int type) {

        throw new RuntimeException ("publish is disabled - Read only
PeerGroup");

    }

    ...

}
```

Tutorial 2: Simple RendezVous PeerGroup

In this tutorial, LightWeightPeerGroup is used in order to implement a PeerGroup that has a specific PeerGroup advertisement (i.e. its own PeerGroupID) but uses all services from a parent PeerGroup except for the RendezVous Service: new specific instance of the RendezVous Service runs into the RendezVous PeerGroup. This PeerGroup is intended to be extended by services and application. It provides a rendezvous peer and edge peer infrastructure that can be used in order to restrict the scope of an existing peergroup.

```
1   import net.jxta.document.Advertisement;

2   import net.jxta.id.ID;

3   import net.jxta.id.IDFactory;

4   import net.jxta.peergroup.PeerGroup;

5   import net.jxta.peergroup.LightWeightPeerGroup;

6   import net.jxta.peergroup.PeerGroupID;

7   import net.jxta.protocol.PeerGroupAdvertisement;

8   import net.jxta.protocol.RdvAdvertisement;

9   import net.jxta.rendezvous.RendezVousService;

10  import net.jxta.impl.rendezvous.RendezVousServiceImpl;

11  public class SimpleRdvPeerGroup extends LightWeightPeerGroup {

12    private RendezVousServiceImpl rdvService = null;

13    private PeerGroup parentPeerGroup = null;
```

```
14    private static String serviceName = "RdvPeerGroupRPV";
15    public SimpleRdvPeerGroup (PeerGroup parentPeerGroup,
16                               PeerGroupAdvertisement adv) {
17        super (adv);
18        this.parentPeerGroup = parentPeerGroup;
19    }
20    public void init(PeerGroup group,
21                     ID assignedID,
22                     Advertisement implAdv) {
23
24
25
26      super.init (group, assignedID, implAdv);
27      // Create a RendezVousService
28      rdvService = new RendezVousServiceImpl ();
29      // Initialialize the RendezVousService.
30      rdvService.init (this,
31                       PeerGroup.rendezvousClassID,
32                       null);
33      }
34    public int startApp(String[] args) {
35        if (rdvService != null) {
36            return rdvService.startApp (args);
37        } else {
38            return START_AGAIN_PROGRESS;
```

```
39        }
40    }
41   public void stopApp() {
42        if (rdvService != null) {
43            rdvService.stopApp();
44        }
45    }
46    public boolean isRendezvous() {
47        if (rdvService == null) {
48            return false;
49        } else {
50            return rdvService.isRendezVous();
51        }
52    }


53    public RendezVousService getRendezVousService() {
54        return rdvService;
55    }
56 }
```

- **line 15:** The constructor takes a PeerGroupAdvertisement as an argument. This advertisement is provided by the caller and describes the SimpleRdvPeerGroup. The advertisement can be created by the JXTA standard AdvertisementFactory.
- **line 20:** The assignedID of the PeerGroup can be provided by the application, but can also be null.
- **line 28:** Instantiates a new RendezVousService. Note that the JXTA platform implementation of the RendezVousService from the JXTA implementation package. A cleaner method would be to analyze the parent peergroup in order to retrieve the implementation of the RendezVous Service, but that it outside the scope of this tutorial.
- **line 30:** Initializes the new RendezVousService instance within the SimpleRdvPeerGroup context.
- **line 34:** The only thing to do when starting a SimpleRdvPeerGroup is to start the private instance of the RendezVousService.

SimpleRdvPeerGroup does not call the *init* method. Unlike *ReadOnlyPeerGroup*, *SimpleRdvPeerGroup* assumes that the application will invoke *init* before using it. It is preferable to let the application initialize the PeerGroup when initialization is expensive, either in CPU cycles, or resources: the application decides when it is appropriate to initialize.

Tutorial 3: Chat Room PeerGroup

This tutorial is using the *SimpleRdvPeerGroup* in order to implement a PeerGroup dedicated to chat room style of communication.

```
1  import net.jxta.peergroup.PeerGroup;
2  import net.jxta.peergroup.LightWeightPeerGroup;
3  import net.jxta.pipe.PipeService;
4  public class ChatRoomPeerGroup extends SimplePeerGroup {
5    private PeerGroup group = null;
6    private PipeService pipe = null;
7    public PipePeerGroup (PeerGroup group, PeerGroupAdvertisement adv) {
8        super (group, adv);
9        this.group = group;
10
11   }
12   public PipeService getPipeService () {
13       return pipe;
14   }
15    public void init(PeerGroup group,
16                     ID assignedID,
17                     Advertisement implAdv) {
18
19
20
21      super.init (group, assignedID, implAdv);
22      // Create a Pipe Service
```

```
23      pipe = new PipeServiceImpl ();
24      // Initialialize the Pipe Service.
25      pipe.init (this,
26                 PeerGroup.pipeClassID,
27                 null);
28      }
29    public int startApp(String[] args) {
30        if (pipe != null) {
31            return pipe.startApp (args);
32        } else {
33            return START_AGAIN_PROGRESS;
34        }
35    }
36   public void stopApp() {
37        if (pipe != null) {
38            pipe.stopApp();
39        }
40    }
41 }
```

B) Password Topic

JXTA allows applications and services to communicate securely over unsecured networks. To acheive secure communications JXTA requires that the peers be authenticated. This allows each peer to trust the other peers it communicates with.

JXTA J2SE 2.3.1 introduced a significant change to the way in which authentication is performed. The JXTA PSE Membership Service, which is responsible for managing authentications and credentials, no longer automatically requests login information from the user. This change was made to give applications greater control over the user interface for authentication and also to control when authentication occurs.

Authenticating via membership APIs

You can perform the authentication directly in your application with :

```
    PeerGroup p = PeerGroupFactory.newNetPeerGroup();

    MembershipService membership = p.getMembershipService();

    Credential cred = membership.getDefaultCredential();

    if( null == cred ) {

        AuthenticationCredential authCred = new
AuthenticationCredential( p, "StringAuthentication", null );

        StringAuthenticator auth = null;

        try {

            auth = (StringAuthenticator) membership.apply( authCred );

        } catch( Exception failed ) {

            ;

        }

        if( null != auth ) {

            auth.setAuth1_KeyStorePassword( "password".toCharArray
() );

            auth.setAuth2Identity( p.getPeerID() );

            auth.setAuth3_IdentityPassword( "password".toCharArray
() );
```

```
            if( auth.isReadyForJoin() ) {

                membership.join( auth );

            }

        }

    }
```

Or you can get the membership service to show an interactive authenticator (usually a dialog) to be completed by the user:

```
    PeerGroup p = PeerGroupFactory.newNetPeerGroup();

    MembershipService membership = p.getMembershipService();

    Credential cred = membership.getDefaultCredential();

    if( null == cred ) {

        AuthenticationCredential authCred = new
AuthenticationCredential( p, "InteractiveAuthentication", null );

        InteractiveAuthenticator auth = (InteractiveAuthenticator)
membership.apply( authCred );

        if( auth.interact() && auth.isReadyForJoin() ) {

           membership.join( auth );

        }

     }
```

C) Sending Large Messge

When is a Message 'Too Large'?

If your application requires that you send large amounts of information between peers it will become necessary to breakup that information into smaller packets for transmission. Generally any message over 16k in size should be chunked in this manner. Sending messages much larger than this puts strain on relay peers and intermediary hosts (routers, firewalls etc.) and may reduce performance.

What Facilities does JXTA have for chunking my data?

JXTA provides JxtaSocket which is capable of performing chunking of data automatically. This is a point to point socket, there is currently no chunking implementation for JxtaMulticastSocket.

How does JxtaSocket chunk my data?

By default JxtaSocket will transmit up to 16k (16384 bytes) of your data. Any larger than that and the data will be sent in multiple messages. This 16k does not include the size of headers and message routing information. This header data may be from 500-2000 bytes and the final message size may be larger than 16k. Beyond 16k JxtaSocket uses a custom protocol to split you data into chunks and transmit them over the network.

I want bigger messages, can I change the default chunk size?

Yes. You can increase the (or decrease) the chunking size by calling JxtaSocket.setOutputStreamBufferSize(int). You should limit this size to a safe value. The largest practical message size is constrained by the size of a TCP packet, which is 64k, minus all the header information: 64k - 61 bytes TCP header - 2000 byte JXTA message header = 63475 bytes or about 61k The size you supply to JxtaSocket.setOutputStreamBufferSize(int) is not constrained so you should ensure it is no larger than 61k. Also note that you must call this function before you request the OutputPipe from JxtaSocket, after you request the OuputPipe it has no effect.

I want to specify the packet size, not the data size, can this be done?

Sorry, JxtaSocket messages vary with size depending on factors outside your control. As such there is no way to enforce a maximum packet size other than to chooses a small data size.

I'm sending a large message over JxtaSocket, why isn't it getting through?

You need to call flush() to force the JxtaSocket to send the last packet. It holds onto the last one in case there is more data to send.

D) Flushing Advertisement from the Cache Manager

JXTA's discovery service provides interfaces to flush (remove) an advertisement from the local cache manager used by the discovery service. Removing advertisements can be accomplished by either passing the advertisement, of the id of the advertisement of interest to flushAdvertisement. The following exmaples illustrate how an advertisement may be removed:

```
   try {
      discovery.flushAdvertisement(advertisement);
   } catch (IOException io) {
       io.printStackTrace();
   }
```

or

```
   try {
      discovery.flushAdvertisement(pipeAdv.getID().toString(),
discovery.ADV);
   } catch (IOException io) {
       io.printStackTrace();
   }
```

At some point the API allowed passing a null ID to trigger removal of all advertisements of a given type, but the feature has since been disabled. However it can be accomplished as illustrated below:

```
   try {
        Enumeration en = discovery.getLocalAdvertisements
(DiscoveryService.ADV, null, null);
        while (en.hasMoreElements()) {
           discovery.flushAdvertisement( (Advertisement) en.nextElement());
        }
   } catch (IOException io) {
       io.printStackTrace();
   }
```

E) Extendable Advertisement

Advertisement extension occurs in two parts: the creation and manipulation of an advertisement XML, performed via an `Advertisement` class object, and the loading of the advertisement XML on a remote peer.

Creating the advertisement XML

For the most part, advertisements are freeform XML documents and can contain whatever information you would like to include in them. For the purposes of this document, lets create a base advertisement which we would like to extend:

```
<FooAdv>
        <Data> Arbitrary data content </Data>
</FooAdv>
```

Now lets pretend we wanted to extend the above advertisement and add an element containing the size of the data in the advertisement. We might construct an advertisement XML that looked something like:

```
<FooAdv type="BarAdv">
        <Size> Size of data </Size>
        <Data> Arbitrary data content </Data>
</FooAdv>
```

Notice that everything is exactly the same as in the original advertisement with the exception of two things: first, the addition of our `Size` element, and second the addition of a `type` attribute on the root element of the advertisement. The reason for adding this `type` attribute will become clear later when we turn our attention to how these advertisements are loaded on a remote peer.

Creating the FooAdv Advertisement base class

Creating an `Advertisement` extension is relatively simple. The `Advertisement` class for `FooAdv` might look something like this:

```
...

public class FooAdv extends ExtendableAdvertisement  {


    /**
     * Instantiator for this Advertisement type.  This is later used
     * when attempting to load a random document via the
     * AdvertisementFactory class.
     */
    public static class Instantiator implements
AdvertisementFactory.Instantiator {

        /**
         * Tells the AdvertisementFactory what type of Advertisement this
         * Instantiator is for.
         */
        public String getAdvertisementType( ) {
```

```
            return FooAdv.getAdvertisementType();
        }

        /**
         *  Called by the AdvertisementFactory to create a new FooAdv.
         **/
        public Advertisement newInstance( ) {
            return new FooAdv();
        }


        /**
         *  Called by the AdvertisementFactory to create a new FooAdv from
         * an XML document.  This process is described in more detail
         * later in this document.
         */
        public Advertisement newInstance( net.jxta.document.Element root ) {
            return new FooAdv( root );
        }
    };

    /**
    * A constructor used to create a new advertisement
    * with default values.
    */
    public FooAdv() {
        setData(null);
    }

    /**
    * A constructor used to create a new advertisement
    * from an XML document.
    */
    public FooAdv(Element root) {

        ...
        /*
        *  Iterate across all child elements of the root element.
        * For each of these elements, call handleElement(elem).
        */

        ...
    }

    /**
    * This class represents a base type, so we define the
    * base type string in such a way that it cannot be changed
    * by subclasses.
    */
    public final String getBaseAdvType() {
        return getAdvertisementType();
    }

    /**
     *  Returns the identifying type of this Advertisement.
     */
```

```
    public static String getAdvertisementType() {
        return "FooAdv";
    }

    /**
     * This method tells the discovery mechanism (SRDI) how to index your
     * custom advertisement.  The field that you specify here determine what
     * attribute names can be used in DiscoveryService.get*Advertisements().
     */
    public String [] getIndexFields() {
        return new String[] { "Data" };
    }

    /**
     * Returns a document encoded as requested.  Most of the work
     * is already done for us.  Just append our data.
     */
    public Document getDocument( MimeMediaType encodeAs ) {
        StructuredDocument adv = (StructuredDocument) super.getDocument
( encodeAs );
        Document doc;
        Element e;

        e = adv.createElement("Data", getData().toString());
        adv.appendChild(e);

        return adv;
        }

    /*
    * Attempt to interpret an element
    */
    protected boolean handleElement(Element elem) {

        /*
        * If our super class understands the element, there
        * should be no reason for additional processing.
        */

        if (super.handleElement( elem )) {
            return true;
        }

        /*
        * If this element is something that we understand, do something
        * useful with the information, such as calling an appropriate
        * accessor method.  If we understand the element, we return
        * true, indicating that the element has been handled.
        */
        if (elem.getName().equals("Data")) {
            setData(elem);
            return true;
        }

        /* If we got here, we dont know what the element is for */
        return false;
```

```
    }

    public void setData(Element elem) {
        ...
        // Do something useful with the data
        ...
    }

    public Object getData() {
        ...
        // Return the data
        ...
    }


}
```

Creating the BarAdv Advertisement subclass

Creating an `Advertisement` extension is much easier. It will look almost exactly like the class being extended, but will only deal with the additional data elements that are specific to the subclass. It will additionally be able to specify more fields to be used as indexes for discovery. Here is what the subclass `BarAdv` might look like:

```
public class BarAdv extends FooAdv {

    /**
     * Instantiator for this Advertisement type.  This is later used
     * when attempting to load a random document via the
     * AdvertisementFactory class.
     */
    public static class Instantiator implements
AdvertisementFactory.Instantiator {

        /**
         * Tells the AdvertisementFactory what type of Advertisement this
         * Instantiator is for.
         */
        public String getAdvertisementType( ) {
            return BarAdv.getAdvertisementType();
        }

        /**
         *  Called by the AdvertisementFactory to create a new FooAdv.
         **/
        public Advertisement newInstance( ) {
            return new BarAdv();
        }

        /**
         *  Called by the AdvertisementFactory to create a new BarAdv from
         * an XML document.  This process is described in more detail
         * later in this document.
         */
```

```
        public Advertisement newInstance( net.jxta.document.Element root ) {
            return new BarAdv( root );
        }
    };



    /**
    * A constructor used to create a new advertisement
    * with default values.
    */
    public BarAdv() {
    }

    /**
    * A constructor used to create a new advertisement
    * from an XML document.
    */
    public BarAdv(Element root) {
        super(root);
    }

    /**
     *  Returns the identifying type of this Advertisement.
     */
    public static String getAdvertisementType() {
        return "BarAdv";
    }

    /**
     * Lets make the "Size" element indexable as well.
     */
    public String [] getIndexFields() {
        return new String[] { "Data", "Size" };
    }

    /**
     * Returns a document encoded as requested.  Most of the work
     * is already done for us.  Just append our data.
     */
    public Document getDocument( MimeMediaType encodeAs ) {
        StructuredDocument adv = (StructuredDocument) super.getDocument
( encodeAs );

        Document doc;

        Element e;

        e = adv.createElement("Size", getSize().toString());

        adv.appendChild(e);

        return adv;
```

```
    }

    /*
    * Attempt to interpret an element
    */
    protected boolean handleElement(Element elem) {

        /*
        * If our super class understands the element, there
        * should be no reason for additional processing.
        */

        if (super.handleElement( elem )) {
            return true;
        }

        /*
        * Grab and set the size info.
        */
        if (elem.getName().equals("Size")) {
            setSize(elem);
            return true;
        }

        /* If we got here, we dont know what the element is for */
        return false;
    }

    public void setSize(Element elem) {

        ...

        // Do something useful with the size info

        ...

    }

    public Object getSize() {

        ...

        // Return the size data

        ...

    }

}
```

How Advertisement instantiation works

Now that we have our advertisement classes defined, we need to tell the JXTA core classes that they exist. To do this, code such as the following snippet would need to be run for each of the new `Advertisement` classes:

```
AdvertisementFactory.registerAdvertisementInstance(

        FooAdv.getAdvertisementType(),

        new FooAdv.Instantiator());
```

The following is critical to the understanding of how `Advertisement` s are created in the JXTA reference implementation

This tells the `AdvertisementFactory` that when it sees an advertisement XML document that has a root node named `FooAdv`, or if the root element has a `type` attribute with a value of `FooAdv`, that this Instantiator should be used to create the local class from the XML. More precisely:

1. The `AdvertisementFactory` will first check the root element for a type attribute. If it has one, it will check to see if there is a registered advertisement instantiator for that type name.
2. If no type attribute exists on the root node, or if there was no instantiator registered for the type listed, the factory will check to see if there is an instantiator registered under the name of the root element - in this case `FooAdv`.

Why the dual methods for finding an instantiator? Lets take another look at our subclass advertisement XML:

```
<FooAdv type="BarAdv">

        <Size> Size of data </Size>

        <Data> Arbitrary data content </Data>

</FooAdv>
```

Normally, with both the `FooAdv` and `BarAdv` instantiators registered with the `AdvertisementFactory` (as would be done on the local peer), the XML above would be loaded using the `BarAdv` instantiator, since the `type` attribute is set to that value. If however, a remote peer only has the `FooAdv` base class registered, ti would have no understanding of the additional `Size` data, but could still be loaded via the `FooAdv` instantiator. Because the advertisement's root node is named `FooAdv`, this will happen. This allows remote peers to benefit from the subclass' advertisements, even when the peer cannot natively understand the additional data.

F) Is JXTA working?

How do you know that JXTA is working?

There are many issues related to JXTA working. The first is connectivity to a RDV/Relay network. The RDVs are used for both message propagation, addressing, and routing. The Relay is used to relay messages to peers behind firewalls. If you cannot see these peers, you cannot talk with other peers. This is sometimes true even on the same LAN. Generally peers in the same computer do work without general problems as long as there are no port conflicts between peers.

Some companies and also personal firewall software like BlackIce and ZoneAlarms will block ports. You either need to reconfigure your firewall or ensure that you have a relay on http port 80.

So, how do you know you are connected? The key is seeing a RDV, so we test to see if the peer is connected.

This code example waits until you are connected to a rendezvouz.

```
    /** Wait for the peer to connect to a Rendezvous. If there is no rende
after 30 seconds, become one.*/

    public boolean waitForRendezVous(PeerGroup peerGroup) throws Exception {

        RendezVousService rdv = peerGroup.getRendezVousService();

        if (rdv == null) {

            throw new RuntimeException("No Rendezvous Service is
configured");

        }

        if(rdv.isRendezVous()){

            LOG.warn("This Peer is a isRendezVous");

            return true;

        }else{

            LOG.warn("Not a RDV");

        }



        if (rdv.isConnectedToRendezVous() ) {

           return true;

        }
```

```
        LOG.error("Waiting for RendezVous");

        int count = 0;

        while (!rdv.isConnectedToRendezVous()&& !rdv.isRendezVous()){

            LOG.error("Waiting for RendezVous:"+count);

            Thread.currentThread().sleep(1*1000);

            if (++count >= 200){

                LOG.error("Rendezvous not found");

                break;

            }

        }

        return rdv.isConnectedToRendezVous();

    }
```

Here is an alternative using the RDV listener interface. The only issue with this example is that there is no expiration after the timeout.

```
rendezvouz = this.netPeerGroup.getRendezVousService();

rendezvouz.addListener(this);

if (!rendezvouz.isConnectedToRendezVous()) {

    try {

        wait();

    } catch(InterruptedException ex) {}

}

// this is one of the methods in the RDV listener

public synchronized void rendezvousEvent(RendezvousEvent event) {

if ((event.getType() == RendezvousEvent.RDVCONNECT) ||
   (event.getType() == RendezvousEvent.RDVRECONNECT)) {
      notify();

}
```

Checking with your Browser

You can check to see if HTTP is running on a peer by specifying its address and port in your browser. For example the following on your local machine :

```
http://localhost:9701/
```

Should Produce:

```
JXTAHELLO tcp://0:0:0:0:0:0:0:1:51181 tcp://127.0.0.1:9801
urn:jxta:uuid-
59616261646162614A7874615032503358 9D4C1829DE4F638732E28F9416F63003 0 1.1
```

The urn:jxta:uuid-596... is the PeerID[?] in the NetPeerGroup[?]

Use telnet to test the TCP port

You can also use telenet to connect. Make sure that you specify the port number.

Warnings and Errors

You might see the following error and/or warnings:

```
  <ERROR 2005-01-08 16:36:47,141 PeerView::getRouteAdv:1630> no Endpoint
Params

  <WARN  2005-01-08 16:36:47,141
DiscoveryServiceImpl::getRemoteAdvertisements:290> resolver has not started
yet, query discarded.
```

The is caused because the platform is not completely booted or perhaps you are not connected to a RDV. Use the waitForRendezVous () code above to ensure that the PeerGroup is initialized and connected to a RDV and these messages should not occur.

JXTA Network Stability

Network stability is a little different. Each RDV in a peer group is connected to another in a chain. Each peer in the network is connected to a RDV. The peers use their RDV for routing pipes because the RDV know via their chain where all other peers are. If the chain is broken, your RDV may not be able to see a RDV with the peer you want to connect to.

NOTE: The publicly maintained network now uses a safe list of valid RDV. This means that a rouge RDV cannot connect to the public network in the Net Peer Group. This has greatly improved things. However, please not that a sub PeerGroup may encounter a similar instability. Peers acting as RDV should be relatively stable.

This problem can happen when a RDV goes offline. The time it takes to heal the chain is only a few minutes at most, but it can affect a lot of operations for that amount of time. If there is a very unstable RDV that goes up and down every few minutes, the problem can cause the network to appear to not be working. The source is often someone that has created a peer that is acting as a RDV but should not be. They are usually testing and that is why the peer is up

and down. This is something that will not happen in the future because the JXTA team is addressing.

To solve the problem for your testing (and this is a more appropriate solution for those in a test mode) should have their own RDV/Relay peer using a PrivateJxtaNetworks.

G) Checking JXTA versions

JXTA uses standard Jar Manifest techniques to label the version of the jars. Here is some example code:

```
public static void versionInfo(String packageToShow){
        try{
        LOG.info("package: "+packageToShow);
            Package aPackage = Package.getPackage(packageToShow);
            if (aPackage == null){
               LOG.info("Package not loaded in this class loader");
               return;
            }
        LOG.info("    ImplementationTitle: "+aPackage.getImplementationTitle());
        LOG.info("    ImplementationVersion: "+aPackage.getImplementationVersion());
        LOG.info("    ImplementationVendor: "+aPackage.getImplementationVendor());

        LOG.info("    SpecificationTitle: "+aPackage.getSpecificationTitle());
        LOG.info("    SpecificationVersion: "+aPackage.getSpecificationVersion());
        LOG.info("    SpecificationVendor: "+aPackage.getSpecificationVendor());

        LOG.info("isSealed: "+aPackage.isSealed());
        /*
        if (aPackage.getSpecificationVersion()!= null){
        LOG.info("isCompatibleWith jxta 2.0: "+aPackage.isCompatibleWith("2.0"));
        }
         */
        }catch(Exception e){
            LOG.error("Error checking JXTA version",e);
        }


    }

Sample call of the method above.

        versionInfo("net.jxta");
        versionInfo("net.jxta.ext.config");
        versionInfo("net.jxta.ext.rdvpeergroup");
```

H) Advertisement Expiration and Lifetime

Expiration and Life Time

This topic tries to answer several questions about the expiration and life time of advertisements in the J2SE JXTA platform:

- How can I use expiration and life time values other than the default ones for peer group advertisements?
- How can I use expiration and life time values other than the default ones for my own peer advertisement?

Expiration and Life Time Basics

The life time determines how long an advertisement is available in the publisher's cache. The expiration time determines when advertisements expire in receiver's caches.

The expiration time of a particular advertisement given to a receiver will never extend beyond its remaining life time. Depending on the remaining life time, an expiration time given to a receiver may be shortened. If the remaining life time is long enough, the advertisement will stay in the receiver's cache for the full expiration time.

Example: If the advertisement is published with a life time of 1h15 and an expiration time of 1 hour and it is requested 45 minutes later, it will expire from the requestor's cache after 30 minutes. However, if the life time is 2 hours it will remain valid in the requestor's cache for its entire expiration time of 1 hour.

At any time though, the publisher or a receiver may also choose to extend the life and expiration times of advertisements.

Specifying Expiration and Life Time Values

Specification of expiration and life time values should be done using long. There is a common pitfall when specifying expiration and life time by multiplying int values. If the result overflows the int, a publication results in an expired advertisement.

Peer Group Advertisements

In order to use your own values for expiration and life time, you need to avoid the convenience routines newGroup() and publishGroup() of the DiscoveryService. These calls publish the peer group advertisement using the default values (i.e., 2 hours for expiration time and 1 year for lifetime).

But a peer group is just a service and you can use the loadModule() methods in connection with publish() as described below.

```
// Create a new all purpose peergroup.
ModuleImplAdvertisement implAdv = netPeerGroup.getAllPurposePeerGroupImplAdvertisement();
// Create a PeerGroupAdvertisement for the peer group.
PeerGroupAdvertisement  apgAdv =
    (PeerGroupAdvertisement) AdvertisementFactory.newAdvertisement
( PeerGroupAdvertisement.getAdvertisementType() );
apgAdv.setPeerGroupID( aPeerGroupID );
apgAdv.setModuleSpecID( implAdv.getModuleSpecID() );
apgAdv.setName( aPeerGroupName );
apgAdv.setDescription( aPeerGroupDescription );
// Publish the PeerGroupAdvertisement for the peer group in the NetPeerGroup.
netPeerGroupDiscovery.publish( apgAdv,DiscoveryService.GROUP,YOUR_LIFETIME,
YOUR_EXPIRATIONTIME );
// Load the PeerGroup
aPeerGroup = (PeerGroup) netPeerGroup.loadModule( apgAdv.getPeerGroupID( ), implAdv );
```

Please note that the call to publish() has to occur before the call to loadModule(). Otherwise the platform does not associate the peer group advertisement with the module implementation advertisement that is being passed to the loadModule() call.

Own Peer Advertisement

The peer advertisement for your own peer is published during the initialization of the discovery service implementation using an infinite lifetime and the default expiration time of 2 hours. Unless you replace the default discovery service with your own, there is not much you can do.

I) Sending Binary Data through Pipe

Here is a simple example of creating a propagate pipe and sending a message with binary data. I also create the advertisement from scratch, using a hardcoded ID called "myPipeID".

```
PipeAdvertisement myPipeAdvertisement = (PipeAdvertisement)
AdvertisementFactory.newAdvertisement(PipeAdvertisement.getAdvertisementType());
myPipeAdvertisement.setPipeID(myPipeID);
myPipeAdvertisement.setType(PipeService.PropagateType);
myPipeAdvertisement.setName("My Pipe");
myPipeAdvertisement.setDescription("A Propagate pipe.");
OutputPipe propagationOutputPipe =
myPeerGroup.getPipeService().createOutputPipe(myPipeAdvertisement, 1000);

Message message = new Message();
byte[] rawBytes = "Here is some bianry data.".getBytes();
MessageElement element = new ByteArrayMessageElement("raw
data",null,rawBytes,null);
message.addMessageElement(element);
propagationOutputPipe.send(message)
```

J) Creating Sub PeerGroup

1) Creating a ModuleSpecID and Creating a PeerGroupID

The following code can be used to create a ModuleSpecID?

Two things to note about this code is that it prints a ModuleSpecID?. The reason for this is that ModuleSpecID? can only be created as a random ID at this time. The reason for this is that it is associated with the implementation and not associated with a dynamic network feature like a PeerGroupID?. ModuleSpecID? is used to associate an implementation and is primarily used to in the creation of a PeerGroup Advertisement.

```
/*
 * CreateModuleSpecID.java
 *
 * Created on December 22, 2004, 4:37 PM
 */

package com.cluck.util;

import net.jxta.id.ID;
import net.jxta.id.IDFactory;
import net.jxta.platform.ModuleClassID;
import net.jxta.platform.ModuleSpecID;
/**
 *
 * @author Daniel Brookshier
 */
public class CreateModuleSpecID{
    public static void main(String[] args){
        System.out.println("To use this ID in the application do the following:");
        System.out.println("ModuleSpecID moduleSpecID = (ModuleSpecID) IDFactory.fromURI( new java.net.URI(\"id
listed below as a string\") );");
        System.out.println(createModuleSpecID().toURI().toString());

    }
    private static ModuleSpecID createModuleSpecID(){
        ModuleClassID classID = IDFactory.newModuleClassID( );
        ModuleSpecID moduleSpecID = IDFactory.newModuleSpecID(classID);
        return moduleSpecID;
    }
}
```

2) Creating a PeerGroup Advertisement

```
public static PeerGroupAdvertisement create(String name, String description, PeerGroup parentGroup,PeerGroupID
    peerGroupID, ModuleSpecID moduleSpecID) {
        try {
```

```
        PeerGroupAdvertisement pga = (PeerGroupAdvertisement)
net.jxta.document.AdvertisementFactory.newAdvertisement( PeerGroupAdvertisement.getAdvertisementType() );

        pga.setName(name);
        pga.setDescription( description );
        pga.setModuleSpecID(moduleSpecID);
        pga.setPeerGroupID( peerGroupID );
        return pga;
      } catch ( Exception e ) {
        LOG.error( "Failed to create group", e );
        return null;
      }
  }
```

3) CreateAPeerGroup

A peer group can be created with the following code. Note that the group is created in the context of the Net Peer Group.

This requires the following prior steps: Creating a ModuleSpecID, Creating a PeerGroupID, and Creating a PeerGroup Advertisement

```
  PeerGroup applicationPeerGroup =  netPeerGroup.newGroup(adv);
```

Another possibility is to create a group based on LightWeightPeerGroup. SimpleRdvPeerGroup? creates a group that inherits the base group functionality and threads, but implements its own RDV to use as a resource in the group.

```
  SimpleRdvPeerGroup applicationPeerGroup = new SimpleRdvPeerGroup(netPeerGroup,adv);
```

Once a group is created, the group must be initialized and started.

```
        applicationPeerGroup.init(netPeerGroup,id,adv);
        applicationPeerGroup.startApp(null);
```

Finally, the group can be published. The platform probably does this for you, but it depends on if you have implemented your own PeerGroup implementation.

```
        applicationPeerGroup.publishGroup( applicationPeerGroup.getPeerGroupName(),
applicationPeerGroup.getPeerGroupName() );
```

K) Informationabout Discovering Pipe and other Advertisements

Introduction

First, the discovery service is a publish and search pattern. At this time there are no events other than an event to tell a peer it has discovered an advertisement it was told to look for.

Second, a search is only going to succeed when the advertisement exists. In other words, you must publish before searching and there is no way to wait for a published advertisement.

With a regular publish, the advertisement remains stored on your peer and an index is created on the RDV. When other peers search, they are actually looking at the RDV index. The RDV then negotiates to allow you to pull the data from the peer. This is of course a very shallow description. The guts of it are complicated, but this is how many people can understand it.

A remote publish of an advertisement moves the advertisement to another peer for storage. This is really no different than a normal publish except there is now a copy of the advert on two peers. Why do this? Imagine you are on a laptop, you want a piece of information in the network that needs to persist while you run off to the airport. Another use is for data that lives a long time and must be accessible even when the original publisher is offline. Another is to balance the load by putting the same data in multiple places.

Using advertisements as part of your JXTA application is not for the faint of heart. There is a lot of thinking and design required to ensure you have a reliable and efficient system. The easiest and best use is for long term network available information. Try to create advertisements that have data that is good for weeks or months at a time, not minutes or hours. Also, at least with the 2.3.4 version of JXTA and until further notice, avoid wildcard queries (see below).

Using DiscoveryService to publish and discover PipeAdvertisements

First is the advertisement. It contains an ID and a name. You can search for both of these. The problem is that if you search by name, and there are lot of adverts with this name, you only get one or two, not all. So unless you can guarantee that there is only one advert of that name, you don't know what ID you are going to get. The catch? The ID is what specifies the pipe. Normally this means that you have a low probability of connecting to a pipe by name that is directly proportional to the number of pipes with that name.

Now, assume we have a name that is unique, how do other peers discover it? The search. But there is a problem, how often should they search? What is the interval? Less than a minute is prrobably a waste of CPU and network resources. Greater than a minute is too slow and users will complain. People actually start getting uppity after a few seconds. Publish and search is not a good way to discover resources - especially pipes by name.

But back to pipes discovered by name for just a second. Remember I said that you need a unique name to stay out of trouble when looking for a specific peer pipe. Ok, that begs for a little thought. If it needs to be unique, I have two problems. If I say it can be a random name, how do I discover it? Really I can't, even with a wild card which then further screws me up because I am back to multiple hit possibilities.

If I say instead that it is uniquely attributed to the one peer, why do I need to discover it? This come to a subject that a lot of people know I have as a passion: Well-known-ID

How do peers know an advertisement has been published?

Common Problem:

I have 3 peers (A, B, C). Peer A creates a Input Pipe and publish the advertisement with a remotePublish in a message. The message arrives to the Rendevous peer that propagates it to the peer A and B. How do peer A and B understand the arrive of message? Is there an interface or listener to tell me when the advertisement arrives?

The answer:

Sorry, this scenario is not possible with DiscoveryService[2]. Many people believe the idea of publishing is closely related to events, but this is not what the system is all about. Discovery in JXTA terms is really about advertising a resource in the network that will be discovered by someone purposefully looking for it.

Second issue, the advertisement does not really leave the peer in most cases when you publish. Just an index hash of the advertisement. The primary reason is that the RDV is used to locate advertisements. If we stored all the content for advertisements, RDV would need to be high powered servers rather than modest PCs.

The other problem is scale. Think about publishing an advertisement that millions are interested in. I need to have some way to get this info to every one if we use a notification model. If we are interested in pipe adverts, that is worse because we assume perhaps that each of these peers has its own pipe open and thus has published a pipe advertisement. This is a many to many situation where all million peers get notified by the other million-1 peers. If we look at just three peers, no big deal, but you can see it can get out of control if notification is included in the pattern.

L) Creating Well known Ids

What is a Well Known ID?
Normally JXTA uses randomly generated IDs for peers, groups, pipes, services and codats. Randomly generated IDs are the best choice for most resouces. A well known ID is an ID who's value is not random, it's value is something which can be calculated or determined by every peer.

What are Well Known IDs normally used for?

Well Known IDs are normally used for resources which are shared by all peers within a group. Examples include common group services, propagate pipes that every peer listens upon and sometimes for the IDs of peers themselves.

Are there alternatives to Well Known IDs?

Yes, you can generate a random ID and then embedd it into your application. By including the ID within the application then every peer will refer to the same resource.

Are there problems with Well Known IDs?

Yes, the biggest problem is the chance for collisions. When choosing Well Known IDs or a scheme for calculating IDs it is very important to ensure that the IDs for different resources have different values. If two resources have the same ID then applications and JXTA will be unable to refer to them properly.

The UUID implementation, for example, will simply copy the first 16 bytes of the byte[] into the ID. The BinaryID[?] which has an address space of 255 bytes which is large, but can have collisions if yu are not careful.

When this interface was first introduced (creating an ID with a byte arra) there was a lot of naive use of it for constructing IDs based on very non-unique source data such as text strings. (and unfortunately some of this is in the JXTA core). As a result the canonical space of these IDs is very small. (less than 28 bits in the case of the peerview advertising pipe id for example). You can also use the BinaryID[?] which has an address space of 255 bytes.

Avoiding ID collisions

It is very important to make sure that the data provided to this ID constructor be well chosen. For generating ids based on most input data sources this means that you should generate a cryptographic hash (SHA1, MD5, etc) of the "well known" data and pass that as the input to the byte[] constructor varient rather than the data itself.

With planning, collisions can be exactly the same as with the auto generated random ID. I use a hash which results in low collisions. If you are worried about collisions, you can also use BinaryID[?] which is a larger ID. Combined with a large hash, you get very low collision probability.

Note that each peer group is a new address space. So a pipe with the same minor ID is different and will not collide in another group.

That said, you do need planning. If you let a user pick their own data for the hash, you get a probable collision because for the most part, people think alike. You also get a Homer-space problem with people picking obvious stuff like 'myAddress' (Doh!). You should also not depend upon names which can collide easily (yes, Mr Smith, I mean you).

But what's good? Choose things that already do not collide. ISBN numbers, phone numbers, email addresses, complete postal addresses, etc. Think about the world around you and find places where the collision is avoided for you. If you are brave you can use passports, drivers license, and social security numbers.

But if you are still afraid of collisions, use a central authority server/database or LDAP. You can still operate in a

P2P world even if you use a server for the first 20 milliseconds to get a unique ID from a database. In fact, many applications in the commercial world need a server to handle the commerce for registering a P2P application. At that time you can get your well-known ID.

You can also use a single address to apply to many peers. If you specify the peer's ID when connecting to a pipe, you only connect to that peer even when there are other peers waiting on that pipe ID. Note: PeerID[2] is different in each PeerGroup you are a member of.

Playing it Safe

Don't assume an ID is collision proof. Accidents can happen and hash algorythums do not guarentee against collisions, they just make them improbable. Even with highly random ID's there is a chance. So, if it can happen, you need to detect it. The first message sent over a pipe should be some piece of data that identifies the target(in the case of bidirectional). If you combine this strategy with PKI and wrap the data in an encrypted envelope, you get authentication too.

A few links for examples and other information

Here are some links talking about Well-known-ID.

WellKnownIDs (this wiki)

If you already know about a P2P service, is that bad?

Power point on JXTA by Daniel Brookshier including well-known-ID

The Socket API in JXTA 2.0 by Daniel Brookshier

Calculating a Well Known ID

The best approach to calculating a Well Known ID is to use some form of hash function upon the resource who's ID is being calculated. SHA1 or MD5 hashes are long enough and random enough to generate good IDs. There is also a utility class in platform. Here is an example:

```
   DigestTool digestTool = new DigestTool();

   PeerGroupBinaryID peerGroupBinaryID = digestTool.createPeerGroupID(netPeerGroup.getPeerGroupID(),
"hello world application", "chat");
```

Here is a useful class that can be used to create standard UUID with a hash. It also includes a way to create a pipe ID.

```
/*

 * com.cluck.jxta.socketChat.MD5ID.java

 * This class is used to create MD5-based UUID

 * Created on November 15, 2002, 5:37 PM

 */

package com.cluck.util;
```

```
import java.io.*;

import java.security.*;

import net.jxta.impl.id.UUID.*;

import net.jxta.id.*;

/**

 *

 * @author  Daniel Brookshier

 */

public class MD5ID {

    private  static final org.apache.log4j.Logger LOG = org.apache.log4j.Logger.getLogger(MD5ID.class.getName
());

    public static final String functionSeperator = "~";



    /**

     * Create a PipeID based on the BinaryID type with a digest of the clearTextID and function.

     *

     * @param peerGroupID Parent peer group ID.

     * @param clearTextID String used as the significant part of the address

     * @param function String used to diferentiate different clearTextID addresses (can be null).

     * @return PipeBinaryID with the digest hash of the string: clearTextID+"~"+function.

     */

    public static final net.jxta.pipe.PipeID createPipeID(net.jxta.peergroup.PeerGroupID peerGroupID,String
clearTextID, String function){

        LOG.info("Creatig pipe ID = peerGroupID:"+peerGroupID+", clearText:'"+clearTextID+"' ,
function:'"+function+"'");

        byte[] digest = generateHash(clearTextID, function);
```

```
        return (net.jxta.pipe.PipeID)net.jxta.id.IDFactory.newPipeID(peerGroupID, digest );

    }



    /**

     * Create a PeerGroupID based on the BinaryID type with a digest of the clearTextID and function.

     *

     * @param peerGroupID Parent peer group ID.

     * @param clearTextID String used as the significant part of the address

     * @param function String used to diferentiate different clearTextID addresses (can be null).

     * @return PeerGroupBinaryID with the digest hash of the string: clearTextID+"~"+function.

     */

    public static final net.jxta.peergroup.PeerGroupID createPeerGroupID(net.jxta.peergroup.PeerGroupID
parentPeerGroupID,String clearTextID, String function){

        LOG.info("Creating peer group ID = peerGroupID:"+parentPeerGroupID+", clearText:"'+clearTextID+"' ,
function:"'+function+"'");

        byte[] digest = generateHash(clearTextID, function);

        //net.jxta.impl.id.UUID.PeerGroupID pg = (net.jxta.impl.id.UUID.PeerGroupID)parentPeerGroupID;

        System.out.println("parentPeerGroupID:"+parentPeerGroupID.getClass().getName());

        net.jxta.peergroup.PeerGroupID peerGroupID = IDFactory.newPeerGroupID( parentPeerGroupID, digest );

        //net.jxta.peergroup.PeerGroupID peerGroupID = new net.jxta.impl.id.UUID.PeerGroupID
((net.jxta.impl.id.UUID.PeerGroupID)parentPeerGroupID,digest);

        return peerGroupID;

    }

    public static final net.jxta.peergroup.PeerGroupID createInftrastructurePeerGroupID(String clearTextID, String
function){

        LOG.info("Creating peer group ID =  clearText:"'+clearTextID+"' , function:"'+function+"'");

        byte[] digest = generateHash(clearTextID, function);
```

```
        net.jxta.peergroup.PeerGroupID peerGroupID = IDFactory.newPeerGroupID(  digest );

        return peerGroupID;

    }

    /**

     * Generates an SHA-1 digest hash of the string: clearTextID+"-"+function or: clearTextID if function was
blank.<p>

     *

     * Note that the SHA-1 used only creates a 20 byte hash.<p>

     *

     * @param clearTextID A string that is to be hashed. This can be any string used for hashing or hiding data.

     * @param function A function related to the clearTextID string. This is used to create a hash associated with
clearTextID so that it is a uique code.

     *

     * @return array of bytes containing the hash of the string: clearTextID+"-"+function or clearTextID if function
was blank. Can return null if SHA-1 does not exist on platform.

     */

    public static final byte[] generateHash(String clearTextID, String function) {

        String id;



        if (function == null) {

            id = clearTextID;

        } else {

            id = clearTextID + functionSeperator + function;

        }

        byte[] buffer = id.getBytes();
```

```
        MessageDigest algorithm = null;


        try {
            algorithm = MessageDigest.getInstance("MD5");
        } catch (Exception e) {
            LOG.error("Cannot load selected Digest Hash implementation",e);
            return null;
        }



        // Generate the digest.
        algorithm.reset();
        algorithm.update(buffer);


        try{
            byte[] digest1 = algorithm.digest();
            return digest1;
        }catch(Exception de){
            LOG.error("Failed to creat a digest.",de);
            return null;
        }
    }
}
```

M) JXTA wiki

For JXTA wiki on Java.net, check http://wiki.java.net/bin/view/Jxta/JxtaFAQ

N) Additional tool - Etheral

The excellent open-source packet sniffer and analyser, Ethereal, provides very thorough support for JXTA. Ethereal can be very useful for both learning JXTA and for debugging application behaviour. Watching what's happening on your network can be incredibly instructive. If you've never used a packet sniffer before then you are sure to learn a lot, and not just about JXTA.

O)

Chapter 10: References

The following Web pages contain information on Project JXTA:

- http://www.jxta.org — home Web page for Project JXTA
- http://spec.jxta.org — Project JXTA specification
- *http://platform.jxta.org* — Project JXTA platform infrastructure and protocols for the J2SE platform binding
- http://platform.jxta.org/java/api/overview-tree.html — public API (Javadoc software)
- http://www.jxta.org/Tutorials.html — numerous Java tutorials

There are numerous technical white papers posted on http://www.jxta.org/white_papers.html. Those of particular interest to developers include:

- *Project JXTA: An Open, Innovative Collaboration*, Sun Microsystems white paper.
- *Project JXTA: A Technology Overview*, Li Gong, Sun Microsystems white paper.
- *Project JXTA Technology: Creating Connected Communities*, Sun Microsystems white paper.
- *Project JXTA Virtual Network*, Bernard Traversat et al., Sun Microsystems white paper.
- Project JXTA: A Loosely-Consistent DHT Rendezvous Walker, Bernard Traversat, Mohamed Abdelaziz, and Eric Pouyoul, Sun Microsystems white paper.
- Introduction to the JXTA Abstraction Layer, Neelakanth Nadgir and Jerome Verbeke, Sun Microsystems.
- PKI Security for JXTA Overlay Networks, Jeffrey Eric Altman, IAM Consulting.

Glossary

Advertisement

Project 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).

Credential

A token used to uniquely identify the sender of a message; can be used to provide message authorization.

Endpoint

See *Peer Endpoint* and *Pipe Endpoint*.

ERP

Endpoint Routing Protocol; used by peers to find routes to other peers.

Gateway

See *Relay Peer*.

Input Pipe

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

J2SE

Java 2 Platform, Standard Edition software.

Message

The basic unit of data exchange between peers; each message contains an ordered sequence of named sub-sections, called message elements, which can hold any form of data. Messages are exchanged by the Pipe Service and the Endpoint Service.

Message Element

A named and typed component of a message (i.e., a name/value pair).

Module

An abstraction used to represent any piece of "code" used to implement a behavior in the JXTA world. Network services are the mode common example of behavior that can be instantiated on a peer.

Module Class

Represents an expected behavior and an expected binding to support the module; is used primarily to advertise the existence of a behavior.

Module Implementation

The implementation of a given module specification; there may be multiple module implementations for a given module specification.

Module Specification

Describes a specification of a given module class; it is one approach to providing the functionality that a module class implies. There can be multiple module specifications for a given module class. The module specification is primarily used to access a module.

NAT

Network Address Translation. Network Address Translation allows a single device, such as a router, to act as an agent between the Internet (or “public network”) and a local (or “private”) network.

Output Pipe

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

P2P

Peer-to-peer; a decentralized networking paradigm in which distributed nodes, or peers, communicate and work collaboratively to provide services.

PBP

Peer Binding Protocol; used by peers to establish a virtual communication channel, or pipe, between one or more peers.

PDP

Peer Discovery Protocol; used by peers to discover resources from other peers.

Peer

Any networked device that implements one or more of the JXTA protocols.

Peer Endpoint

A URI that uniquely identifies a peer network interface (e.g., a TCP port and associated IP address).

Peer Group

A collection of peers that have a common set of interests and have agreed upon a common set of services.

Peer Group ID

ID that uniquely identifies a peer group.

Peer ID

ID that uniquely identifies a peer.

PIP

Peer Information Protocol; used by peers to obtain status information (uptime, state, recent traffic, etc.) from other peers.

Pipe

An asynchronous and unidirectional message transfer mechanism used by peers to send and receive messages; pipes are bound to specific peer endpoints, such as a TCP port and associated IP address.

Pipe Endpoint

Pipe endpoints are referred to as *input pipes* and *output pipes*; they are bound to peer endpoints at runtime.

PKI

Public Key Infrastructure. Supports digital signatures and other public key-enabled security services.

PRP

Peer Resolver Protocol; used by peers to send generic queries to other peer services and receive replies.

Relay Peer

Maintains information on routes to other peers, and helps relay messages to peers. (Previously referred to as router peer.)

Rendezvous Peer

Maintains a cache of advertisements and forwards discovery requests to other rendezvous peers to help peers discover resources.

RVP

Rendezvous Protocol; responsible for propagating messages within a peer group.

TLS

Transport Layer Security. (See *http://www.ietf.org/html.charters/tls-charter.html* for more details.)

URI

Uniform Resource Identifier. A compact string of characters for identifying an abstract or physical resource. (See *http://www.w3.org/Addressing/URL/ URI_Overview.html* for more details.)

URN

Uniform Resource Name. A kind of URI that provides persistent identifiers for information resources. (See IETF RFC 2141, *http://www.ietf.org/rfc/rfc2141.txt*, for more details.)

Troubleshooting

This appendix discusses commonly encountered problems compiling and running JXTA applications.

Errors compiling JXTA applications

Check that your are including the correct `jxta.jar` file in your compilation statement (`-classpath` option). If you have downloaded multiple versions, verify that you are including the most recent version in your compilation statement.

Note – The required `.jar` files can be downloaded from the JXTA Web site:
http://download.jxta.org.

Errors running JXTA applications

Setting the classpath variable

When you run your JXTA application, you need to set the `-classpath` variable to indicate the location of the required `.jar` files. Be sure to include the same version that you used when compiling your JXTA application. Although you need only the `jxta.jar` file for compilation, you need multiple `.jar` files when running a JXTA application.

Note – See on page for a list of the required Java `.jar` files.

Unable to discover JXTA peers

If you are unable to discover other JXTA resources (peers, peer groups, or other advertisements), you may have configured your JXTA environment incorrectly. Common configuration issues include the following:

- If you are located behind a firewall or NAT, you must use HTTP and specify a relay node.
- If you are using TCP with NAT, you may need to specify your NAT public address.
- You may need to specify at least one rendezvous node.

Remove the JXTA configuration file (`PlatformConfig`) and then re-run your application. When the JXTA Configurator window appears, enter your configuration information. See Appendix , for more details on running the JXTA Configurator.

Using the JXTA Shell

You can use the JXTA Shell to help troubleshoot configuration issues and test JXTA services. Commands are available to discover JXTA advertisements, create JXTA resources (e.g., groups, pipes, messages, and advertisements), join and leave peer groups, send and receive messages on a pipe, and much more.

For example, to verify correct network configuration you can use the JXTA Shell command "rdvstatus" to display information about your current rendezvous status (i.e., if you are configured as a rendezvous peer, and who your current rendezvous peers are). You can also use "search -r" to send out discovery requests, and then use "peers" to display any peers that have been discovered — to confirm that network connectivity is working as expected.

For more information on downloading and using the JXTA Shell, please see:

http://shell.jxta.org/

Starting from a clean state

Some problems can be caused by stale configuration or cache information. Try removing the JXTA configuration files and cache directory:

n `./.jxta/PlatformConfig`

n `./.jxta/cm` (directory)

Re-launch the application. When the Configuration window appears, enter the appropriate information for your network configuration. See *http://platform.jxta.org/java/confighelp.html* for more details on running the JXTA Configurator.

Displaying additional log information

If your JXTA application isn't behaving as you expect, you can turn on additional logging so that more information is displayed when your application runs.

To select a new logging, or trace, level, re-run the JXTA Configurator and from the Advanced Settings tab select the desired Trace Level from the pull-down menu. The default trace level is *error; warn*, *info*, and *debug* levels provide more information. For more information on running the JXTA Configurator, please see Appendix , Emphasisparatextefault .

You can also choose to edit the `PlatformConfig` file in the current directory rather than re-running the JXTA Configurator. For example, the following entry in `PlatformConfig` sets the trace level to "warning":

```
<Dbg>
   warn
<\Dbg>
```

Removing User name or Password

The first time you run a JXTA application, you will be prompted to enter a user name and password. Each subsequent time you run the application, you will be prompted to enter the same user name and password pair. If you forget either the user name or the password, you can remove the `cm` directory (located in the under $JXTAHOME directory, by default .jxta) and then re-run the application. The JXTA Configurator will be displayed, and you can enter a new user name and password. See http://wiki.java.net/bin/view/Jxta/WebHome for more details on running the JXTA Configurator.