JXSE v2.6
The JXTA Java™ Standard Edition Implementation

Programmer's Guide

by Jérôme Verstrynge

July, 2010

Trademarks

- Sun, Sun Microsystems, the Sun Logo, and Java are trademarks, registered trademarks, or service marks of Sun Microsystems, Inc. in the United States and other countries.
- All SPARC trademarks are used under license and are trademarks or registered trademarks of SPARC International, Inc., in the United States and other countries. Products bearing SPARC trademarks are based upon an architecture developed by Sun Microsystems, Inc.
- ORACLE is a registered trademark in the United States and other countries, of Oracle Corporation.
- Berkeley DB is a registered trademark in the United States and other countries, of Oracle Corporation.
- UNIX is a registered trademark in the United States and other countries, exclusively licensed through X/Open Company, Ltd.
- JXTA is a registered trademark in the United States and other countries, of Sun Microsystems, Inc.

Please Recycle

Table of Contents

Introduction

This document is a follow-up on the JXSE 2.5 Programmers Guide. Its purpose is to describe the new features which have been made available as part of the 2.6 release of the JXTA implementation in the Java Programming language (JXSE). This release is also known as JXSE 2.6. It also clarifies some concepts introduced in the JXTA/JXSE 2.5 Programmers Guide and provides additional, updated or new information where necessary.

The basics of JXTA can be found in the JXSE 2.5 Programmer's Guide.

Acknowledgments

This document was written with direct and indirect contributions from Mike (Bondolo) Duigou, Iain McGinniss Lorenzo Bigagli, Valerio Angelini, Mike Cummings and Simon Temple.

I would also like to thanks all those who contributed code or other software engineering skills and knowledge to the 2.6 release since release 2.5 (in alphabetical order)[1]:

- Adrian Ivan
- Ariel Ro
- Ben St. Pierre
- Bernd (neunzert)
- Cees Pieters
- Daniel Brookshier
- Iain McGinniss
- James Todd
- John Boyle
- Lorenzo Bigagli
- Mike Cumings
- Mike Duigou
- Mirron Rozanov
- Mohamed Hamada
- Qiang Zeng
- Simon Temple
- Valerio Angelini
- William (Bill) Middleton

Highlights

- *OSGi* – JXSE can now be started as an OSGi bundle/service to access the NetworkManager. The Apache Felix OSGi implementation has been integrated in JXSE and can be used as the standard OSGi framework.
- *Configuration Objects* – A new set of configuration objects have been created as part of the introduction of OSGi in JXSE. These can easily be loaded or saved as XML documents, or via input and output.
- *Task Manager* – In order to reduce the consumption of Threads and other resources, a single task manager object has been implemented to use a single thread group. The code has been updated to use centralize task requests and thread consumption.
- *Cache Manager* – A new implementation of the cache manager based on Apache Derby and H2 is now available. These can replace the current standard b-tree file system implementation.
- *SRDI* – An in-memory implementation of the SRDI is now available.
- *Maven* – Three pom.xml files and a proper directory structure were existing code can be copied has been implemented.

1 Let me know if I forgot someone.

- *New TCP Implementation* – A new implementation of the TCP layer based on the NIO Netty has been implemented.

Update

Latest guide information, updates and errata will be made available on our wiki[2].

2 See http://kenai.com/projects/jxse/pages/Home

What Is New?

This section introduces the new features implemented in the 2.6 release, in alphabetic order.

API Brush-Up

It is generally agreed that an API addresses the required functionality in a software application. It should be a framework guiding the implementation of these without restraining technical implementation choices.

When it comes to the JXTA/JXSE project, the API should reflect functionality described or implied by the JXTA Specification document. It should also cover all use cases (i.e., ways of using the application) making it practical to operate.

Unfortunately, the current status of JXTA/JXSE API does not fully meet these requirements . Some API code references implementation details (e.g. the *impl* packages). Some implementation code is also present in the API, while some implementation 'concepts' deserve to be taken up in the API. The API documentation is sometimes too technical or overly related to implementation code or concepts. The API is not always easy to learn, to use, to maintain or flexible enough to evolve. It sometimes leaves too many doors open for interpretation.

There are situations where code behavior is unpredictable. Some users have to read implementation code to understand the behavior of the API. In some cases (for example, query messages), users have to refer to implementation code, which against the idea that they should only refer to the API for their own applications.

The design could also be improved in order to avoid having the user implement what the API module can also do (for example XML document manipulation) and reduce the need for boilerplate code.

API Objects referencing Impl code
`net.jxta.platform.NetworkConfigurator`
`net.jxta.platform.NetworkManager`
`net.jxta.peergroup.NetPeerGroupFactory`
`net.jxta.socket.JxtaSocket`
`net.jxta.socket.JxtaMulticastSocket`
`net.jxta.util.JxtaBiDiPipe`
`net.jxta.util.JxtaServerPipe`

The API brush-up initiative aims at making the code more consistent and clean. There has been quite some debate about the necessity of this process in the JXTA community. Some questioned the utility of it, others raised concerns regarding the downwards compatibility with existing applications. Some were concerned that modifying an API can alienate users, while others argued that the opposite happens when performed properly. A survey of the JXTA community showed that apparently, no one has actually implemented their own version of the delivered JXTA API objects[3].

Performing an API brush-up is a complex task which can only happen over several releases. Some raised the possibility of creating a new API from scratch. Others intuition that this should happen through the integration of the OSGi framework. The debate is still open, but the effort has been started for release 2.6.

Some obsolete and already deprecated code has been removed from the API. Other code (such as methods for example), have been deprecated or created as necessary. More methods and classes will be deprecated and removed, or created, in future releases.

When API (or Impl) code is removed, it is first deprecated for at least one release version, before being definitely removed (or refactored). Where necessary, the Javadoc is updated to guide users towards alternative or new code. The corresponding issues are well documented in the issue tracker.

Instantiator

A new object called `net.jxse.JxseInstantiator` has been implemented in the API. The purpose of this class is to provide a mechanism to load `Class` object instances from their fully qualified name, for later

3 Or if they did, they did not communicate this during the survey.

instantiation using reflection. It can be used to instantiate implementation classes from within the API without having the API code referring to the implementation packages code.

The instantiator could be used in a future release to remove references to the implementation code from within the API. The Instantiator is already used to instantiate the OSGi framework.

JXTA vs JXSE Packages

Currently, the code is organized in `net.jxta` and `net.jxta.impl` packages. However, new `net.jxse` packages have been introduced in order to start separating the JXTA concepts (i.e., from the specifications) from their actual implementation.

Configuration Objects

A new set of configuration objects have been implemented to easily load and save network configurations in collaboration with the OSGi framework:

Level	Package & Class
API	net.jxta.configuration.JxtaConfiguration
API	net.jxta.configuration.JxtaConfigurationException
API	net.jxta.configuration.JxtaPeerConfiguration
API	net.jxta.configuration.JxtaTransportConfiguration
API	net.jxse.configuration.JxseHttpTransportConfiguration
API	net.jxse.configuration.JxseMulticastTransportConfiguration
API	net.jxse.configuration.JxseTcpTransportConfiguration
API	net.jxse.configuration.JxsePeerConfiguration

- `JxtaConfiguration` – This object represents the concept of configurations in JXTA implementations. It specifically inherits of the Java `Properties` object, which offers facilities to load and save registered properties either via InputStreams and OutputStreams, or via XML files.

  By default, the Properties object does not load / save default values for properties. The `JxtaConfiguration` object has been enhanced to handle this issue.
- `JxtaConfigurationException` – This object represents a generic JXTA configuration issue.

  `JxtaPeerConfiguration` – This object inherits of `JxtaConfiguration` and represents the JXTA peer configuration concept.

  `JxtaTransportConfiguration` – This abstract object inherits of `JxtaConfiguration` and represents the JXTA transport configuration concept.
- `JxseHttpTransportConfiguration` – This object implements the JxtaTransportConfiguration object for the JXSE HTTP transportation layer implementation.

  `JxseMulticastTransportConfiguration` – This object implements the `JxtaTransportConfiguration` object for the JXSE Multicast transportation layer.

  `JxseTcpTransportConfiguration` – This object implements the `JxtaTransportConfiguration` object for the JXSE TCP transportation layer.
- `JxsePeerConfiguration` – This object implements the JxtaPeerConfiguration object for JXSE. It contains an intance of `JxseHttpTransportConfiguration`, `JxseMulticastTransportConfiguration` and `JxseTcpTransportConfiguration` that will be saved and restored automatically when it is saved and loaded. This object can easily be updated to handle future transportation protocols.

Until JXTA/JXSE 2.5, it was the responsibility of the NetworkManager and of the NetworkConfigurator to save and retrieve existing configuration. The new configuration objects provide an alternative way of setting and getting peer configurations. It is the responsibility of the user to load and save the JxsePeerConfiguration objects properly.

In the OSGi section, we will describe how to obtain a completely configured instance of the NetworkManager (and corresponding NetworkConfigurator) from a JxsePeerConfiguration object.

Connectivity

Several methods and features have been implemented in this release to facilitate the monitoring of connectivity of peers and peergroups. Some buggy and functionally useless methods have been deprecated too.

RendezVous Service Methods

Some rendezvous service methods have been modified as following, new methods have been added too:

Method	Comment
`connectToRendezVous(...)`	Deprecated since 2.5 – Connection to RDVs is managed by core code.
`disconnectFromRendezVous(...)`	Deprecated since 2.6 – Connection to RDVs is managed by core code.
`getConnectedRendezVous()`	Deprecated since 2.6 – This method contains bugs. It returns the list of EDGE connected to this peer when it is a RDV.connection to RDVs is managed by core code.
`getDisconnectedRendezVous()`	Deprecated since 2.5 – Connection to RDVs is managed by core code.
`getConnectedPeers()`	Deprecated since 2.6 – This method is functionally useless, since it does not provide the list of EDGE peer connected to this peer when it is a RDV.
`getConnectedPeerIDs()`	Deprecated since 2.6 – This method is functionally useless, since it does not provide the list of EDGE peer IDs connected to this peer when it is a RDV.
`getLocalWalkView()`	Deprecated since 2.5 – Method contains bugs.
`challengeRendezVous(...)`	Deprecated since 2.6 – Connection to RDVs is managed by core code.
`getLocalRendezVousView()`	New in 2.6 – Provides the list of RDV an EDGE is connected to, or the list of RDVs a RDV knows through its peer view.
`getLocalEdgeView()`	New in 2.6 – Provides the list of EDGES a RDV is connected to.

The deprecated methods will be removed in the next release.

Method `isConnectedToRendezVous()`

In earlier releases, the values returned by the `isConnectedToRendezVous()` method were ambiguous. It was false for RDVs, true or false for EDGE peers and true for ADHOC peers. In release 2.6, the behavior of this method changes:

- ADHOC – Such peers typically does not participate in peer group activities and does not establish lease connection to rendezvous, although they will propagate group messages whenever possible. They only use (LAN) multicasting as a means of communication.

  As from release 2.6, a call to the `isConnectedToRendezVous()` method on their `RendezvousService` always returns `false`, instead of `true` before. This is consistent with the fact that ADHOC peers do not interact with infrastructure peers.

- EDGE – Such peers try (desperately) to connect and to remain connected to one and only rendezvous peer in their peer group. If 'their' rendezvous fails, they will automatically try to reconnect to another rendezvous (and may become one if autostart is enabled). They use all means of communication available.

  A call to the `isConnectedToRendezVous()` method on the `RendezvousService` always returns `true` or `false`, to indicate whether they are connected to a rendezvous or not.

- RENDEZVOUS – These peers accept lease connections from EDGE peers and connect to other rendezvous as possible. They use all means of communication available.

  As from release 2.6, a call to the `isConnectedToRendezVous()` method on their `RendezvousService` will return `true` if they are connected to other RDV peers, via their peerview, or `false` otherwise.

Some applications using older versions of JXTA/JXSE rely on the fact that the `isConnectedToRendezVous()` method returns true for ADHOC peers, in order to find out whether they were ready to perform peergroup activities. The motivation behind this is not clear, but these applications will have to update their code.

Endpoint Service Methods

A couple of methods related to connectivity have been added to the rendezvous service.

Connection To Relay Peers

Two new methods have been implemented regarding relay connectivity:

- `isConnectedToRelayPeer()` – This method indicates whether an EDGE peer is connected to a RendezVous. Otherwise, it always returns false.
- `getConnectedRelayPeers()` – This method returns a collection of the relay peer IDs this peer is connected to.

Ping – Is Reachable?

In release 2.5, the `EndpointService` contained a deprecated `ping` method. Usually, when a ping is performed, it is followed by some communication with the remove device.

A new method called `IsReachable(PeerID pid, boolean tryToConnect)` has been added to the `EndpointService`, and indicates whether the target peer is reachable.

- If `tryToConnect=false`, the method returns `true` if an active connection has already been established to the target peer, else it returns `false`.
- If `tryToConnect=true`, the method returns `true` if an active connection has already been established to the target peer, else it tries to establish a connection to the target peer. If this connection is successful, the method returns `true`, else it returns `false`.

Assuming that a peer is reachable, a `Messenger` can be retrieved from one of the `getMessenger()` methods in the `EndpointService` to send messages.

EndpointRouter & Route Controller

One additional method has been implemented in the endpoint service to retrieve the endpoint router, that is, the object capable of dealing with unresolved endpoint addresses (typically, for peers from which we only know about its ID). These objects implement the endpoint routing protocol.

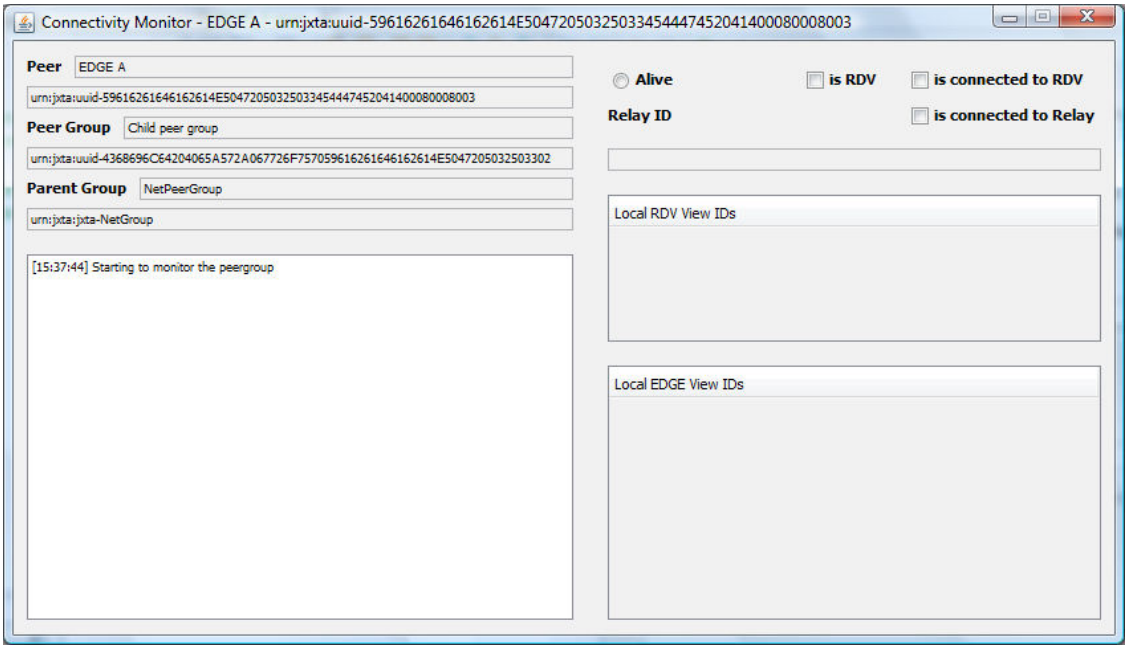
- `getEndpointRouter()` – This method returns an `EndpointRoutingTransport` containing a special method on top of traditional transport methods: `getRouteController()`.

`RouteController` objects handle routes to target peers. They offer five API methods:

- `addRoute(RouteAdvertisement newRoute)` – This method can be used to provide route hints to the system. The route controller will check them and return an integer code indicating the result of the operation (ok, already exists, failed, opened direct route, invalid route).
- `getAllRoutes()` – Returns all known routes.
- `getRoutes(PeerID inPID)` – Returns all known routes to a target peer.
- `getLocalPeerRoute()` – Returns a route to this peer.
- `isConnected(PeerID pid)` – Returns true when a direct route to a target peer is known.
- These endpoint service methods, together with the new rendezvous methods, can help users to test and debug connectivity issues in their applications.

Connectivity Code Example

The following pages provide new tutorial code examples to monitor connectivity status between peers. A NetPeerGroup and a child peer group is created on each peer.



The examples refer to two other classes: `ConnectivityMonitor` and `DelayedJxtaNetworkStopper`. The code is available in the appendix.

EDGE A

```
/**
 * Simple EDGE peer connecting via the NetPeerGroup.
 */
public class EDGE_A {

    // Static

    public static final String Name_EDGE_A = "EDGE A";
    public static final PeerID PID_EDGE_A = IDFactory.newPeerID(
        PeerGroupID.defaultNetPeerGroupID, Name_EDGE_A.getBytes());
    public static final int TcpPort_EDGE_A = 9710;
    public static final File ConfigurationFile_EDGE_A =
        new File("." + System.getProperty("file.separator") + Name_EDGE_A);

    public static final String ChildPeerGroupName = "Child peer group";
    public static final PeerGroupID ChildPeerGroupID =
        IDFactory.newPeerGroupID(PeerGroupID.defaultNetPeerGroupID, ChildPeerGroupName.getBytes());

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

        try {

            // Removing verbose
            // Logger.getLogger("net.jxta").setLevel(Level.WARNING);
```

```
Logger.getLogger(EdgePeerRdvService.class.getName()).setLevel(Level.CONFIG);

// Removing any existing configuration?
NetworkManager.RecursiveDelete(ConfigurationFile_EDGE_A);

// Creation of the network manager
final NetworkManager MyNetworkManager = new NetworkManager(
        NetworkManager.ConfigMode.EDGE,
        Name_EDGE_A, ConfigurationFile_EDGE_A.toURI());

// Retrieving the network configurator
NetworkConfigurator MyNetworkConfigurator = MyNetworkManager.getConfigurator();

// Setting Configuration
MyNetworkConfigurator.setTcpPort(TcpPort_EDGE_A);
MyNetworkConfigurator.setTcpEnabled(true);
MyNetworkConfigurator.setTcpIncoming(true);
MyNetworkConfigurator.setTcpOutgoing(true);
MyNetworkConfigurator.setUseMulticast(true);

// Setting the Peer ID
MyNetworkConfigurator.setPeerID(PID_EDGE_A);

// Adding RDV a as a seed
MyNetworkConfigurator.clearRendezvousSeeds();

String TheSeed = "tcp://" + InetAddress.getLocalHost().getHostAddress() + ":"
        + RENDEZVOUS_A.TcpPort_RDV_A;
URI LocalRendezVousSeedURI = URI.create(TheSeed);
MyNetworkConfigurator.addSeedRendezvous(LocalRendezVousSeedURI);

TheSeed = "tcp://" + InetAddress.getLocalHost().getHostAddress() + ":"
        + RENDEZVOUS_B.TcpPort_RDV_B;
LocalRendezVousSeedURI = URI.create(TheSeed);
MyNetworkConfigurator.addSeedRendezvous(LocalRendezVousSeedURI);

// Starting the JXTA network
PeerGroup NetPeerGroup = MyNetworkManager.startNetwork();

// Starting the connectivity monitor
new ConnectivityMonitor(NetPeerGroup);

// Disabling any rendezvous autostart
NetPeerGroup.getRendezVousService().setAutoStart(false);

// Retrieving a module implementation advertisement
ModuleImplAdvertisement TheModuleImplementationAdvertisement = null;

try {
    TheModuleImplementationAdvertisement =
        NetPeerGroup.getAllPurposePeerGroupImplAdvertisement();
} catch (Exception ex) {
    System.err.println(ex.toString());
}

// Creating a child group
PeerGroup ChildPeerGroup = NetPeerGroup.newGroup(
        ChildPeerGroupID,
        TheModuleImplementationAdvertisement,
        ChildPeerGroupName,
        "For test purposes..."
        );
```

```
            if (Module.START_OK != ChildPeerGroup.startApp(new String[0]))
                System.err.println("Cannot start child peergroup");

            new ConnectivityMonitor(ChildPeerGroup);

            // Stopping the network asynchronously
            ConnectivityMonitor.TheExecutor.schedule(
                new DelayedJxtaNetworkStopper(
                    MyNetworkManager,
                    "Click to stop " + Name_EDGE_A,
                    "Stop"),
                0,
                TimeUnit.SECONDS);

        } catch (IOException Ex) {

            System.err.println(Ex.toString());

        } catch (PeerGroupException Ex) {

            System.err.println(Ex.toString());

        }

    }

}
```

RENDEZVOUS A

```
/**
 * Simple RENDEZVOUS peer connecting via the NetPeerGroup.
 */
public class RENDEZVOUS_A {

    // Static

    public static final String Name_RDV_A = "RENDEZVOUS A";
    public static final PeerID PID_RDV_A =
        IDFactory.newPeerID(PeerGroupID.defaultNetPeerGroupID, Name_RDV_A.getBytes());
    public static final int TcpPort_RDV_A = 9711;
    public static final File ConfigurationFile_RDV_A =
        new File("." + System.getProperty("file.separator") + Name_RDV_A);

    public static final String ChildPeerGroupName = "Child peer group";
    public static final PeerGroupID ChildPeerGroupID =
        IDFactory.newPeerGroupID(PeerGroupID.defaultNetPeerGroupID, ChildPeerGroupName.getBytes());

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

        try {

            // Removing verbose
            Logger.getLogger("net.jxta").setLevel(Level.WARNING);

            // Removing any existing configuration?
            NetworkManager.RecursiveDelete(ConfigurationFile_RDV_A);
```

```
            // Creation of the network manager
            final NetworkManager MyNetworkManager = new NetworkManager(
                    NetworkManager.ConfigMode.RENDEZVOUS,
                    Name_RDV_A, ConfigurationFile_RDV_A.toURI());

            // Retrieving the network configurator
            NetworkConfigurator MyNetworkConfigurator = MyNetworkManager.getConfigurator();

            // Setting Configuration
            MyNetworkConfigurator.setTcpPort(TcpPort_RDV_A);
            MyNetworkConfigurator.setTcpEnabled(true);
            MyNetworkConfigurator.setTcpIncoming(true);
            MyNetworkConfigurator.setTcpOutgoing(true);
            MyNetworkConfigurator.setUseMulticast(true);

            // Setting the Peer ID
            MyNetworkConfigurator.setPeerID(PID_RDV_A);

            // Starting the JXTA network
            PeerGroup NetPeerGroup = MyNetworkManager.startNetwork();

            // Starting the connectivity monitor
            new ConnectivityMonitor(NetPeerGroup);

            // Retrieving a module implementation advertisement
            ModuleImplAdvertisement TheModuleImplementationAdvertisement = null;

            try {
                TheModuleImplementationAdvertisement =
                    NetPeerGroup.getAllPurposePeerGroupImplAdvertisement();
            } catch (Exception ex) {
                System.err.println(ex.toString());
            }

            // Creating a child group
            PeerGroup ChildPeerGroup = NetPeerGroup.newGroup(
                    ChildPeerGroupID,
                    TheModuleImplementationAdvertisement,
                    ChildPeerGroupName,
                    "For test purposes..."
                    );

            if (Module.START_OK != ChildPeerGroup.startApp(new String[0]))
                System.err.println("Cannot start child peergroup");

            // Enable rendezvous
            ChildPeerGroup.getRendezVousService().startRendezVous();

            new ConnectivityMonitor(ChildPeerGroup);

            // Stopping the network asynchronously
            ConnectivityMonitor.TheExecutor.schedule(
                new DelayedJxtaNetworkStopper(
                    MyNetworkManager,
                    "Click to stop " + Name_RDV_A,
                    "Stop"),
                0,
                TimeUnit.SECONDS);

        } catch (IOException Ex) {

            System.err.println(Ex.toString());
```

```
        } catch (PeerGroupException Ex) {

            System.err.println(Ex.toString());

        }

    }

}
```

Content Management

Shortly after Jxta 2.5 was released, a new means was implemented to exchange contents (for example files) between peers. It allows 'serving' peers to offer content to 'client' peers. The following pages describe how to use the content manager.

Client Side

```
package tutorial.content;

import net.jxta.content.ContentID;
import net.jxta.content.ContentService;
import net.jxta.content.ContentTransfer;
import net.jxta.content.ContentTransferEvent;
import net.jxta.content.ContentTransferListener;
import net.jxta.content.ContentTransferAggregator;
import net.jxta.content.ContentTransferAggregatorEvent;
import net.jxta.content.ContentTransferAggregatorListener;
import net.jxta.content.TransferException;
import net.jxta.id.IDFactory;
import net.jxta.peergroup.PeerGroup;
import net.jxta.platform.NetworkManager;
import java.io.*;
import java.net.*;

/**
 * This tutorial illustrates the use of the Content API from the
 * perspective of the client.
 * <p/>
 * The client will attempt to locate and retrieve a Content instance
 * identified by a specified ContentID.  Once the retrieval completes,
 * the client will exit.
 */
public class ContentClient {

    private transient NetworkManager manager = null;

    private transient PeerGroup netPeerGroup = null;
    private transient boolean waitForRendezvous = false;
    private final ContentID contentID;

    /**
     * Content transfer listener used to receive asynchronous updates regarding
     * the transfer in progress.
     */
    private ContentTransferListener xferListener =
            new ContentTransferListener() {
        public void contentLocationStateUpdated(ContentTransferEvent event) {
            System.out.println("Transfer location state: "
                    + event.getSourceLocationState());
            System.out.println("Transfer location count: "
                    + event.getSourceLocationCount());
```

```
        }

        public void contentTransferStateUpdated(ContentTransferEvent event) {
            System.out.println("Transfer state: " + event.getTransferState());
        }

        public void contentTransferProgress(ContentTransferEvent event) {
            Long bytesReceived = event.getBytesReceived();
            Long bytesTotal = event.getBytesTotal();
            System.out.println("Transfer progress: "
                    + bytesReceived + " / " + bytesTotal);
        }
    };

    /**
     * Content transfer aggregator listener used to detect transitions
     * between multiple ContentProviders.
     */
    private ContentTransferAggregatorListener aggListener =
            new ContentTransferAggregatorListener() {
        public void selectedContentTransfer(ContentTransferAggregatorEvent event) {
            System.out.println("Selected ContentTransfer: "
                    + event.getDelegateContentTransfer());
        }

        public void updatedContentTransferList(ContentTransferAggregatorEvent event) {
            ContentTransferAggregator aggregator =
                event.getContentTransferAggregator();
            System.out.println("ContentTransfer list updated:");
            for (ContentTransfer xfer : aggregator.getContentTransferList()) {
                System.out.println("    " + xfer);
            }
        }
    };

    /**
     * Constructor.
     *
     * @param id ContentID of the Content we are going to attempt to retrieve.
     * @param waitForRendezvous true to wait for rdv connection, false otherwise
     */
    public ContentClient(ContentID id, boolean waitForRendezvous) {
        contentID = id;

            // Start the JXTA network
        try {
            manager = new NetworkManager(
                    NetworkManager.ConfigMode.ADHOC, "ContentClient",
                    new File(new File(".cache"), "ContentClient").toURI());
            manager.startNetwork();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }
        netPeerGroup = manager.getNetPeerGroup();
        if (waitForRendezvous) {
            manager.waitForRendezvousConnection(0);
        }
    }

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

```
    public void run() {
        try {
            /*
             * Get the PeerGroup's ContentService instance.
             */
            ContentService service = netPeerGroup.getContentService();

            System.out.println("Initiating Content transfer");
            // Create a transfer instance
            ContentTransfer transfer = service.retrieveContent(contentID);
            if (transfer == null) {
                /*
                 * This can happen if no ContentProvider implementations
                 * have the ability to locate and retrieve this ContentID.
                 * In most scenarios, this wouldn't happen.
                 */
                System.out.println("Could not retrieve Content");
            } else {
                /*
                 * Add a listener so that we can watch the state transitions
                 * as they occur.
                 */
                transfer.addContentTransferListener(xferListener);

                /*
                 * If the transfer is actually an aggregation fronting multiple
                 * provider implementations, we can listen in on what the
                 * aggregator is doing.
                 */
                if (transfer instanceof ContentTransferAggregator) {
                    ContentTransferAggregator aggregator = (ContentTransferAggregator) transfer;
                    aggregator.addContentTransferAggregatorListener(aggListener);
                }

                /*
                 * This step is not required but is here to illustrate the
                 * possibility.  This advises the ContentProviders to
                 * try to find places to retrieve the Content from but will
                 * not actually start transferring data.  We'll sleep after
                 * we initiate source location to allow this to proceed
                 * outwith actual retrieval attempts.
                 */
                System.out.println("Starting source location");
                transfer.startSourceLocation();
                System.out.println("Waiting for 5 seconds to demonstrate source location...");
                Thread.sleep(5000);

                /*
                 * Now we'll start the transfer in earnest.  If we had chosen
                 * not to explicitly request source location as we did above,
                 * this would implicitly locate sources and start the transfer
                 * as soon as enough sources were found.
                 */
                transfer.startTransfer(new File("content"));

                /*
                 * Finally, we wait for transfer completion or failure.
                 */
                transfer.waitFor();
            }
        } catch (TransferException transx) {
            transx.printStackTrace(System.err);
        } catch (InterruptedException intx) {
```

```
            System.out.println("Interrupted");
        } finally {
            stop();
        }
    }

    /**
     * If the java property RDVWAIT set to true then this demo
     * will wait until a rendezvous connection is established before
     * initiating a connection
     *
     * @param args none recognized.
     */
    public static void main(String args[]) {

        /*
         System.setProperty("net.jxta.logging.Logging", "FINEST");
         System.setProperty("net.jxta.level", "FINEST");
         System.setProperty("java.util.logging.config.file", "logging.properties");
         */
        if (args.length > 1) {
            System.err.println("USAGE: ContentClient [ContentID]");
            System.exit(1);
        }
        try {
            Thread.currentThread().setName(ContentClient.class.getName() + ".main()");
            URI uri;
            if (args.length == 0) {
                uri = new URI(ContentServer.DEFAULT_CONTENT_ID);
            } else {
                uri = new URI(args[0]);
            }
            ContentID id = (ContentID) IDFactory.fromURI(uri);
            String value = System.getProperty("RDVWAIT", "false");
            boolean waitForRendezvous = Boolean.valueOf(value);
            ContentClient socEx = new ContentClient(id, waitForRendezvous);
            socEx.run();
        } catch (Throwable e) {
            System.out.flush();
            System.err.println("Failed : " + e);
            e.printStackTrace(System.err);
            System.exit(-1);
        }
    }

    private void stop() {
        manager.stopNetwork();
    }

}
```

Server Side

```
package tutorial.content;

import net.jxta.content.Content;
import net.jxta.content.ContentID;
import net.jxta.content.ContentProviderEvent;
import net.jxta.content.ContentProviderListener;
import net.jxta.content.ContentService;
import net.jxta.content.ContentShare;
import net.jxta.content.ContentShareEvent;
```

```
import net.jxta.content.ContentShareListener;
import net.jxta.discovery.DiscoveryService;
import net.jxta.document.FileDocument;
import net.jxta.document.MimeMediaType;
import net.jxta.id.IDFactory;
import net.jxta.peergroup.PeerGroup;
import net.jxta.platform.NetworkManager;
import net.jxta.protocol.ContentShareAdvertisement;
import java.io.*;
import java.util.*;
import java.net.*;

/**
 * This tutorial illustrates the use Content API from the
 * perspective of the serving peer.
 * <p/>
 * The server is started and instructed to serve a file
 * to all peers.
 */
public class ContentServer {

    static final String DEFAULT_CONTENT_ID =
        "urn:jxta:uuid-59616261646162614E50472050325033901EA80A652D476C9D1089545CEDE7B007";

    private transient NetworkManager manager = null;
    private transient PeerGroup netPeerGroup = null;
    private transient boolean waitForRendezvous = false;
    private final ContentID contentID;
    private final File file;

    /**
     * Content provider listener used to be notified of any activity being
     * performed by the contrnt provider.
     */
    private ContentProviderListener provListener =
            new ContentProviderListener() {
        public void contentShared(ContentProviderEvent event) {
            logEvent("Content shared:", event);
        }

        public void contentUnshared(ContentProviderEvent event) {
            logEvent("Content unshared: ", event);
        }

        public boolean contentSharesFound(ContentProviderEvent event) {
            throw new UnsupportedOperationException("Not supported yet.");
        }
    };

    /**
     * Content share listener used to be notified of any activity during
     * a content transfer.
     */
    private ContentShareListener shareListener = new ContentShareListener() {
        public void shareSessionOpened(ContentShareEvent event) {
            logEvent("Session opened: ", event);
        }

        public void shareSessionClosed(ContentShareEvent event) {
            logEvent("Session closed", event);
        }

        public void shareAccessed(ContentShareEvent event) {
```

```
            logEvent("Share access", event);
        }
    };

    /**
     * Constructor.
     *
     * @param toServe file to serve
     * @param id ContentID to use when serving the toServer file, or
     *  {@code null} to generate a random ContentID
     * @param waitForRendezvous true to wait for rdv connection, false otherwise
     */
    public ContentServer(File toServe, ContentID id, boolean waitForRendezvous) {
        try {
            manager = new NetworkManager(
                    NetworkManager.ConfigMode.ADHOC, "ContentClient",
                    new File(new File(".cache"), "ContentClient").toURI());
            manager.startNetwork();
        } catch (Exception e) {
            e.printStackTrace();
            System.exit(-1);
        }

        netPeerGroup = manager.getNetPeerGroup();
        if (waitForRendezvous) {
            manager.waitForRendezvousConnection(0);
        }
        file = toServe;
        if (id == null) {
            contentID = IDFactory.newContentID(
                    netPeerGroup.getPeerGroupID(), false);
        } else {
            contentID = id;
        }
    }

    /**
     * Interact with the server.
     */
    public void run() {
        try {
            /*
             * Get the PeerGroup's ContentService instance.
             */
            ContentService service = netPeerGroup.getContentService();

            /*
             * Here we setup a Content object that we plan on sharing.
             */
            System.out.println("Creating Content object");
            System.out.println("   ID     : " + contentID);
            FileDocument fileDoc = new FileDocument(file, MimeMediaType.AOS);
            Content content = new Content(contentID, null, fileDoc);
            System.out.println("   Content: " + content);

            /*
             * Now we'll ask the ContentProvider implementations to share
             * the Content object we just created.  This single action may
             * result in more than one way to share the object, so we get
             * back a list of ContentShare objects.  Pragmatically, we
             * are likely to get one ContentShare per ContentProvider
             * implementation, though this isn't necessarily true.
             */
```

```
            List<ContentShare> shares = service.shareContent(content);

            /*
             * Now that we have some shares, we can advertise them so that
             * other peers can access them.  In this tutorial we are using
             * the DiscoveryService to publish the advertisements for the
             * ContentClient program to be able to discover them.
             */
            DiscoveryService discoService = netPeerGroup.getDiscoveryService();
            for (ContentShare share : shares) {
                /*
                 * We'll attach a listener to the ContentShare so that we
                 * can see any activity relating to it.
                 */
                share.addContentShareListener(shareListener);

                /*
                 * Each ContentShare has it's own Advertisement, so we publish
                 * them all.
                 */
                ContentShareAdvertisement adv = share.getContentShareAdvertisement();
                discoService.publish(adv);
            }

            /*
             * Wait forever, allowing peers to retrieve the shared Content
             * until we terminate.
             */
            System.out.println("Waiting for clients...");
            synchronized(this) {
                try {
                    wait();
                } catch (InterruptedException intx) {
                    System.out.println("Interrupted");
                }
            }
            System.out.println("Exiting");
        } catch (IOException io) {
            io.printStackTrace();
        } finally {
            stop();
        }
    }

    /**
     * If the java property RDVWAIT set to true then this demo
     * will wait until a rendezvous connection is established before
     * initiating a connection
     *
     * @param args none recognized.
     */
    public static void main(String args[]) {

        /*
         System.setProperty("net.jxta.logging.Logging", "FINEST");
         System.setProperty("net.jxta.level", "FINEST");
         System.setProperty("java.util.logging.config.file", "logging.properties");
         */
        if (args.length > 2) {
            System.err.println("USAGE: ContentServer [File] [ContentID]");
            System.exit(1);
        }
```

```
        try {
            File file;
            if (args.length > 0) {
                file = new File(args[0]);
                // Use the file specified
                if (!file.exists()) {
                    System.err.println("ERROR: File '" + args[0] + "' does not exist");
                    System.exit(-1);
                }
            } else {
                // Create and use a temporary file
                file = File.createTempFile("ContentServer_", ".tmp");
                FileWriter fileWriter = new FileWriter(file);
                fileWriter.write("This is some test data for our demonstration Content");
                fileWriter.close();
            }

            Thread.currentThread().setName(ContentServer.class.getName() + ".main()");
            URI uri;
            if (args.length == 2) {
                uri = new URI(args[1]);
            } else {
                uri = new URI(DEFAULT_CONTENT_ID);
            }
            ContentID id = (ContentID) IDFactory.fromURI(uri);
            String value = System.getProperty("RDVWAIT", "false");
            boolean waitForRendezvous = Boolean.valueOf(value);
            ContentServer socEx = new ContentServer(file, id, waitForRendezvous);
            socEx.run();
        } catch (Throwable e) {
            System.out.flush();
            System.err.println("Failed : " + e);
            e.printStackTrace(System.err);
            System.exit(-1);
        }
    }

    ////////////////////////////////////////////////////////////////////////
    // Private methods:

    private void logEvent(String title, ContentProviderEvent event) {
        System.out.println("ContentProviderEvent " + title);
        System.out.println("        ContentID: " + event.getContentID());
        System.out.println("        Provider : " + event.getContentProvider());
        System.out.println("        Shares   : " + event.getContentShares().size());
        for (ContentShare share : event.getContentShares()) {
            System.out.println("            " + share.toString());
        }
    }

    private void logEvent(String title, ContentShareEvent event) {
        System.out.println("ContentShareEvent - " + title);
        System.out.println("        Source    : " + event.getContentShare());
        System.out.println("        Name      : " + event.getRemoteName());
        System.out.println("        Data Start: " + event.getDataStart());
        System.out.println("        Data Size : " + event.getDataSize());
    }

    private void stop() {
        manager.stopNetwork();
    }

}
```

Metering Code

The metering project (https://jxse-metering.dev.java.net/) and the corresponding code exist in the JXTA/JXSE project since 2003. The purpose of this code is to provide a framework to collect metering data from peers running within peergroups.

Until JXTA/JXSE 2.5, the corresponding code was generated by the `Ant` build file (via the `-meterCheck` and `-meterConfig` targets) if it could not be found within the project. This code was not part of the Subversion repository used by the JXTA community.

It has been decided to include the RUNTIME generation version of this code in the Subversion repository for the sake of simplicity; it prevents unnecessary pre-processing at compile time.

Generated Code & Configuration

The generated code was the following pairs of files (for the RUNTIME generation):

File type	File name
Conditional build	net\jxta\impl\endpoint\transportMeter\TransportMeterBuildSettings.java
Runtime conditional build	net\jxta\impl\endpoint\transportMeter\ConditionalTransportMeterBuildSettings.java
Conditional build	net\jxta\impl\meter\MeterBuildSettings.java
Runtime conditional build	net\jxta\impl\meter\ConditionalMeterBuildSettings.java
Conditional build	net\jxta\impl\endpoint\endpointMeter\EndpointMeterBuildSettings.java
Runtime conditional build	net\jxta\impl\endpoint\endpointMeter\ConditionalEndpointMeterBuildSettings.java
Conditional build	net\jxta\impl\resolver\resolverMeter\ResolverMeterBuildSettings.java
Runtime conditional build	net\jxta\impl\resolver\resolverMeter\ConditionalResolverMeterBuildSettings.java
Conditional build	net\jxta\impl\rendezvous\rendezvousMeter\RendezvousMeterBuildSettings.java
Runtime conditional build	net\jxta\impl\rendezvous\rendezvousMeter\ConditionalRendezvousMeterBuildSettings.java

Typically, the conditional build file (for example `TransportMeterBuildSettings.java`) defines a boolean as following:

```
...
public static final boolean TRANSPORT_METERING =
   ConditionalTransportMeterBuildSettings.isRuntimeMetering();
...
```

This boolean is used in many places in the code to check whether transport metering code should be executed or not.

The corresponding runtime file (in this case `ConditionalTransportMeterBuildSettings.java`), defines the following `isRuntimeMetering()` static method:

```
...
public static boolean isRuntimeMetering() {
```

```
  boolean runtimeMetering = false;
  try {
    ResourceBundle userResourceBundle =
        ResourceBundle.getBundle("net.jxta.user");
    String meteringProperty = "net.jxta.meter.conditionalTransportMetering";
    String meteringValue = userResourceBundle.getString(meteringProperty);
    runtimeMetering = "on".equalsIgnoreCase(meteringValue);
  } catch (Exception ignored) {
  }
  return runtimeMetering;
}
...
```

By default, this method returns false, unless a metering property can be found in the `net.jxta.user` property file bundled as part of the jxta.jar file. In this case, the value returned by the method depends of value configured in that file. If it is set to "ON", the method will return true and the corresponding static boolean will be set to true too. Else, the method returns false.

Five properties which can be configured to enable metering code:

Boolean	Property
TRANSPORT_METERING	net.jxta.meter.conditionalTransportMetering
METERING	net.jxta.meter.conditionalMetering
ENDPOINT_METERING	net.jxta.meter.conditionalEndpointMetering
RESOLVER_METERING	net.jxta.meter.conditionalResolverMetering
RENDEZVOUS_METERING	net.jxta.meter.conditionalRendezvousMetering

One can replicate the other configuration (always ON, always OFF) which were available as part of metering code generation by either setting all properties to ON or OFF.

There is always the possibility of regenerating metering code with the `meterConfig` ant target.

Optimization & Dead Code Removal

Some may be concerned by the bytecode size increase of the metering code in the final `jxse.jar`. We are relying on the fact that the metering booleans are static final. This implies that at runtime/loadtime, the JIT (Just-In-Time) Java bytecode compiler can rely on the value of these booleans to perform dead code removal optimizations where possible.

By default, the metering booleans are set to false and the corresponding code will be optimized at runtime.

NetworkConfigurator Setters & Getters

In JXSE/JXTA 2.5, the NetworkConfigurator object offers several setter methods, but only few getter methods. A new set of getter methods has been implemented to match the set of setter methods.

New Implementations of Cache Manager & SRDI Index

JXTA-JXSE makes use an entity called the Cache Manager to store the details of advertisements received and published across the P2P network, and to allow these advertisements to be searched on "indexed" properties using a pattern matching method.

Prior to JXTA-JXSE 2.6, only a single implementation of the cache manager existed, which resides in `net.jxta.impl.cm.Cm`. This implementation utilised a custom file system Btree.

1. The implementation used an excessive number of file handles. Once the store reached a certain size, and depending on the operating system, no more file handles could be acquired by the process. This caused random failures either in the cache manager or within other subsystems.
2. The implementation was rather slow, as it was entirely file system based with minimal in-memory caching. As the cache manager is a core component of the system, this approach was adversely affecting system performance.

As a result, it was desirable to create a new implementation, but in such a manner that any implementation could be selected at system startup. The designer should be able to choose this without any reliance on a particular dependency injection framework or module system. This would allow the community time to determine how modules and dependencies should be handled in the future (which is, after all, intended to be a core competency of the JXTA P2P framework).

In addition to the cache manager, the SRDI Index is an equally important but smaller component. This allows rendezvous nodes to store information on where to direct advertisement queries through the network. The SRDI Index was originally also implemented using the custom file system BTree implementation, and has now been replaced for similar reasons.

Cache Manager Implementations

Four implementations of the cache manager are available for JXTA-JXSE:

1. `XIndice` – this is the original cache manager implementation, modified only to fit within the refactored cache manager loading scheme. This is still the default implementation for 2.6, despite it's known issues, to allow for seamless upgrading to 2.6.
2. `H2` – an implementation using the H2 embedded database. This is the new preferred implementation, as it is significantly faster, and has proven to be much more stable in large systems.
3. `Derby` – an implementation using the Apache Derby embedded database, which ships with Sun's JavaSE distribution. This is significantly slower than the H2 implementation, due to what appears to be Derby's inferior storage mechanism, as compared to H2.
4. `BDB`[4] – an implementation based on Oracle's Berkeley DB Java Edition. This implementation is extremely fast, however BDB-JE is not open source and requires a license from Oracle to be used in commercial environments. As such, this is not shipped with 2.6 by default, and must be explicitly imported to be used.

SRDI Index Implementations

Two implementations of the cache manager are available for JXTA-JXSE:

1. `XIndice` – the original implementation, modified only to fit within the refactored SRDI Index loading scheme. This is still the default implementation for 2.6.
2. `BDB` – an implementation based on Oracle's Berkeley DB Java Edition. A commercial license is required for this from Oracle if it is to be used in a commercial product, and so is not shipped with 2.6 by default, and must be explicitly imported to be used.

Selecting a New Implementation

The mechanism used to choose a particular cache manager and srdi index implementation is simple, and uses Java system properties. The system properties are:

- `net.jxta.impl.cm.cache.impl` - The value associated with this system property indicates the cache manager implementation to be used. The legal values for 2.6 are:
  - `XIndice` – `net.jxta.impl.cm.XIndiceAdvertisementCache` [default]
  - `H2` – `net.jxta.impl.cm.sql.H2AdvertisementCache`
  - `Derby` – `net.jxta.impl.cm.sql.DerbyAdvertisementCache`
  - `BDB` – `net.jxta.impl.cm.bdb.BerkeleyDbAdvertisementCache`
- `net.jxta.impl.cm.SrdiIndex.backend.impl` - The value associated with this system property indicates the SRDI index implementation to be used. The legal values for 2.6 are:

4 The corresponding code is not included in core code, but is available form the \contrib directory.

- XIndice – `net.jxta.impl.cm.XIndiceSrdi`
- In-Memory – `net.jxta.impl.cm.InMemorySrdi` [default]
- BDB – `net.jxta.impl.cm.srdi.bdb.BerkeleyDbSrdi`

These system properties can be used to set your preferred mechanism. For instance, if you wished to use the H2 implementation of the cache manager and the in-memory implementation of the SRDI index, you could set the following on the command line:

```
> java \ -Dnet.jxta.impl.cm.cache.impl=net.jxta.impl.cm.sql.H2AdvertisementCache \
-Dnet.jxta.impl.cm.SrdiIndex.backend.impl=net.jxta.impl.cm.InMemorySrdiIndexBackend \
-jar MyJxtaApp.jar
```

OSGi Level 1 Integration

The JXTA community has shown interest in integrating the OSGi framework. A study, available in the appendix, indicates that 3 levels of integration are to be considered. Level 1 is the possibility to start the JXTA network as an OSGi service.

A new set of objects have been implemented:

Level	Package & Class (or object)
API	net.jxta.service.JxtaService
API	net.jxse.service.JxseService
API	net.jxse.service.JxseModule
API	net.jxse.OSGi.JxseOSGi.properties
API	net.jxse.OSGi.JxseOSGiFramework
API	net.jxse.OSGi.JxseOSGiFrameworkLauncher
API	net.jxse.OSGi.Services.JxseOSGiNetworkManagerService
API	net.jxse.OSGi.Services.JxseOSGiService
IMPL	net.jxse.impl.OSGi.Felix.properties
IMPL	net.jxse.impl.OSGi.JxseOSGiFelixFrameworkLauncher
IMPL	net.jxse.impl.OSGi.Activators.JxseOSGiNetworkManagerActivator
IMPL	net.jxse.impl.OSGi.Services.JxseOSGiNetworkManagerServiceImpl

- `JxtaService` – This object represents the service concept in JXTA implementations. It is currently an empty interface and is not integrated with the existing `Service` interface as further work on the future architecture of JXSE is expected.
- `JxseModule` – This interface represents the JXTA module concept in JXSE.
- `JxseService` – This interface represents the JXTA service concept in JXSE. It extends both the `JxtaService` and the `JxseModule` interfaces.
- `JxseOSGi.properties` – This is a property files containing the fully qualified name of the OSGi framework launcher.
- `JxseOSGiFramework` – This object loads the `JxseOSGi.properties` file with static code and instantiates the OSGi framework via the `INSTANCE` public static final attribute, using reflexion. It also implements a static method to retrieve services via a `ServiceTracker`. As soon as this object is referenced in the code execution, the OSGi framework is loaded using the class implementing the `JxseOSGiFrameworkLauncher` interface.
- `JxseOSGiFrameworkLauncher` – This API interface defines a method to retrieve the OSGi framework instance. It must be used as a base for any implementation of JXSE OSGi framework launchers, since it is used in the `JxseOSGiFramework` class. This level of abstraction allows multiple implementations of OSGi framework launchers.

- `JxseOSGiService` – This API abstract class represents the OSGi service concept in JXSE. Typically, if the community decides in the future to implement all JXTA services via OSGi in JXSE, this would have to extend this abstract class.
- `JxseOSGiNetworkManagerService` – This OSGi abstract class implements a service providing access to a NetworkManager instance to start the JXTA network. It provides methods to set and get peer configurations using the new configuration objects.
- `Felix.properties` – The Felix properties files contains properties to configure the Apache Felix OSGi framework implementation. Currently:
  - The bundle cache is flushed on framework initialization
  - Felix does NOT shut down the JVM when stopped
  - The `JxseOSGiNetworkManagerActivator` (described further) is automatically loaded and stopped with the framework
- `JxseOSGiFelixFrameworkLauncher` – This class implements the `JxseOSGiFrameworkLauncher` interface to launch the Apache Felix OSGi framework implementation.
- `JxseOSGiNetworkManagerActivator` – This OSGi bundle/service activator creates and registers the JxseOSGiNetworkManagerService implementation in the OSGi framework.
- `JxseOSGiNetworkManagerServiceImpl` – This class implements the JxseOSGiNetworkManagerService interface.

Using the OSGi Integration

To simply access the OSGi framework without starting the JXTA network:

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

        // Starting the framework
        JxseOSGiFramework.INSTANCE.start();
        ...
        // Accessing the framework services
        ServiceReference[] Services = JxseOSGiFramework.INSTANCE.getRegisteredServices();
        ...
        // Stopping the framewok
        JxseOSGiFramework.INSTANCE.stop();

        // Waiting for the stop (0 = wait indefinitely)
        FrameworkEvent StopEvent = JxseOSGiFramework.INSTANCE.waitForStop(0);
        ...
        System.exit(0);
        ...

}
```

To start the JXTA network using OSGi:

```
…

 // Creating a configuration object
JxsePeerConfiguration JPC = new JxsePeerConfiguration();

 // Setting the configuration
TheNMS.setPeerConfiguration(JPC);
```

```
// Retrieving the NetworkManager
NetworkManager TheNM = null;

try {

    TheNM = TheNMS.getConfiguredNetworkManager();

    TheNM.startNetwork();
    ...
    TheNM.stopNetwork();

} catch (PeerGroupException ex) {
    System.err.println(ex.toString());
} catch (IOException ex) {
    System.err.println(ex.toString());
} catch (Exception ex) {
    System.err.println(ex.toString());
}
…
```

Task Manager

In order to reduce thread and resource consumption, a new task manager object has been implemented. The code has been modified to make sure all the peergroup and services instances submit their tasks to the task manager. Again, this is transparent to the user and no modifications are required in the software applications that have been developed.

TCP Layer Implementation

A new TCP layer communication has been implemented, based on the Netty library[5]. It is bit-wise compatible with the existing implementation and replaces it completely. This is transparent to the user and no modifications are required in the software applications.

A new entry in the `net.jxta.platform.Module` text file of the META-INF.services package has been created:

```
...
urn:jxta:uuid-deadbeefdeafbabafeedbabe000000090106
net.jxta.impl.endpoint.tcp.TcpTransport Reference Implementation of the TCP
Message Transport
...
urn:jxta:uuid-deadbeefdeafbabafeedbabe000000090107
net.jxta.impl.endpoint.netty.NettyTransport Reference Implementation of the
TCP Message Transport
...
```

Reverting to the 'Old' TCP Implementation

If one needs to revert to the old TCP implementation, the 000000090107 line must be modified to point to `net.jxta.impl.endpoint.tcp.TcpTransport` instead of `net.jxta.impl.endpoint.netty.NettyTransport`.

What Goes?

ACL

5 See http://www.jboss.org/netty/.

Before release 2.6, there was a possibility to define access control list (ACL) URIs, which store a set of valid URIs. Corresponding methods have been deprecated and will be removed in a future release because of a functional conflict with 'use only' booleans. These indicate whether peers should only use (or not) seeds and seeding URIs defined in the configuration.

Direct & Canonical Messengers

Until version 2.5, the `EndpointService` offered two methods, `getDirectMessenger(...)` & `getCanonicalMessenger(...)`, in order to retrieve direct and canonical messengers to target peers.

- Direct Messenger – These messengers were implemented to improve communication latency between peers on a LAN. They have no impact on throughput. These messengers default to TCP addresses whenever possible.

  Unfortunately, this causes connectivity issues with peers located behind NATs. If a peer has a private address on a LAN, it cannot be reached using this address from the WAN. Yet, direct messengers are still trying to reach those peers this way and must fail before another route is attempted, causing long connectivity delays.

- Canonical Messenger – Canonical messengers (to a specific peer) are complementary to channel Messenger (to a specific service on a specific peer). As a result, end users are never confronted with the channel messengers, which are conceptually redundant. Whatever can be accomplished with a channel messenger can be accomplished with a canonical messenger as well.

  Moreover, there is no need to make a conceptual distinction between `Messengers` and `CanonicalMessengers` at the end user level.

Consequently, the `getDirectMessenger(...)` and `getCanonicalMessenger(...)` have been deprecated in the `EndpointService`. End users should not rely on these anymore and should use one of the remaining `getMessenger()` methods.

- `GetMessengerImmediate(...)` - Returns a `Messenger` which may or may not be resolved to the target peer. If the messenger needs to be resolved, this will happen asynchronously. The method returns immediately.
- `GetMessenger(...)` - This method returns a `Messenger` to the target peer, but will try to resolve it if necessary. This method blocks until the `Messenger` is resolved.

PROXY Configuration Mode

The PROXY configuration mode is used by JXME applications which do not have/implement the full JXTA functionality. These can connect to PROXY enabled peers in order to reach the entire JXTA network.

Unfortunately, the JXME project is currently incomplete and has not been maintained for several years. Hence, the PROXY configuration code has been deprecated and will be removed in a future release, unless someone is willing to take an active role in this project.

Some Clarifications

This section was created to answer common questions about the JXSE implementation of the JXTA protocols which were not covered in the previous documentation.

Advertisement Lifecycle

The general lifecycle of advertisement is the following:

- *Publish* – A unique ID is added to the advertisements if none is available. Then, it is saved in the local cache manager with an absolute lifetime and a relative expiration. A SRDI entry (containing the key values of the advertisment is created).

  If the peer is an EDGE, the SRDI entry is registered in the delta tracking system. The discovery service is in charge of pushing the entries to the connected rendezvous. It pushes all entries the first time, then only deltas every 30 seconds.

  The rendezvous receiving these entries forward them to one replica peer.

  The advertisement is garbage collected from the cache manager when its lifetime is expired. The SRDI entries are deleted after their expiration.
- *Remote Publish* – A resolver response message is created with the advertisement, and it is propagated to all target peers. When it is received by a remote peer, it is published locally.
- *Flush* – The advertisement is removed from the local cache.

REM: The route resolver and the pipe resolver also forward SRDI entries of route advertisements and pipe advertisements respectively.

Seeds

JXSE distinguishes two types of seeds: rendezvous seeds and relay seeds. A seed is a JXTA route to a peer. Both type of seeds serve different purposes:

- *Rendezvous seeds* – Rendezvous seeds allow peers to connect to RENDEZVOUS enabled peers in their peergroups. This is necessary to receive messages and queries propagated within a peer group. It is also necessary to propagate queries and messages from this peer through the peergroup.

  Typically, it does not make sense to set a peer as a rendezvous seed if it is not reachable from the WAN (in other words, if it is hidden behind a NAT). Only peers having a public IP address should be set as rendezvous seeds.
- *Relay seeds* – Relay seeds help solving connectivity issues for peers hidden behind NATs. Typically, such peers will connect to relay seeds, which make them accessible from the WAN via the relay seed. From time-to-time they check the relay seeds for messages to fetch.

  It does not make sense to set peers as relay seeds if they are not reachable from the WAN too.

It is the responsibility of a peer to be connected to a rendezvous in order to participate in peer group activities.

Lifecycle

When registering seeds in the NetworkConfigurator, the user can specify URIs containing the physical endpoint address of the seed peer or seeding URIs, which are source locations read for route advertisements (or physical endpoint addresses) to seed peers.

Rendezvous seeds are used by `PeerView` and `EdgePeerRdvService`. `PeerView` are used by `EdgePeerRdvService`:

- Peerviews retrieve the seed configuration from the `RdvConfig` advertisement extracted from the peer group configuration itself and creates a `URISeedingManager` object.

  Later a `Kicker` task is created. It uses multiple strategy to propagate this peer with other peers (one or many) of the peerview via peerview messages. Known rendezvous advertisements are also shared with other peers.

When peerview messages are received by this peer, the corresponding rendezvous advertisement can be removed if it is a failure notification or added in the current peerview if it is considered acceptable (validity filters are applied).

- The `EdgePeerRdvService` creates a `URISeedingManager` and adds configured seed and seeding URIs. Later this seeding manager is used when RDV connections are checked by the `MonitorTask` object.

Relay seed configuration is stored in `RelayConfigAdv`. This information is passed by `RelayServer` and `RelayClient` by `RelayTransport`. `RelayServer` do not use seed information. `RelayClient` creates a `URISeedingManager` and adds configured seed and seeding URIs. It is later used by its `run()` method to establish connections to relay peers.

Rendezvous Auto-Start

The JXSE `RendezvousService` allows peers to promote or demote themselves from a RENDEZVOUS status. This means that EDGE and ADHOC peers (for example) can automatically become rendezvous in their peergroup if there is a lack of rendezvous peers, or return to an EDGE mode if there are not enough EDGE peers connected to it.

Some users are concerned that auto-start peers have private IP addresses. It makes sense to have such peers in specific situations, if they are themselves connected to other rendezvous. In order to participate in peergroup activites, other peers can connect to them, one way or the other.. This lowers the burden of rendezvous have a public IP address.

More information about autostart is available in the 'Under the hood' section.

Module & Service Lifecycle

All modules and services in JXSE are implemented via the Module interface, which is extended by the Service interface. The Module interface defines a set of methods that have to be implemented. Software engineers are free to implement these (for customized services for example) according to their needs, but should strive to comply with the following recommended practices for JXSE:

1. After an object instance of a Module has been created, the `init(...)` method should be callable before any other methods, because it will be invoked by calls to `PeerGroup.loadModule(...)` for example. When invoked, the peergroup to which the Module or Service is attached should be provided as a parameter, together with its ModuleClassID (for a peergroup, its ID should also be provided), and eventually an implementation advertisement. The latter can be a module implementation advertisement for a customized module.

   Any errors during initialization should throw a `PeerGroupException`. This can occur when the `init(...)` method is called twice for example, or when it is called after calling `startApp()`. However, it is not absolutely necessary if such calls are harmless.

   Next, the `startApp()` method should be callable:

   - START_OK – This value must be returned if the Module or Service was started successfully.
   - START_AGAIN_PROGRESS or START_AGAIN_STALLED – This value must be returned if the startApp() method should be called again later to finalize the Module or Service start. This is useful when the start of multiple modules depends of each other. Gradual starting procedures can be implemented.
   - START_DISABLED – This value must be returned if the Module or Service cannot be started, whatever the reason.

   A module or service should not be considered as started until it returns START_OK.

   At last, the `stopApp()` method should be called to stop the Module or Service. Many maintain a 'started' boolean to verify its started/stopped status. Typically, the `stopApp()` method should not perform anything when the module is in a stopped status.

The typical lifecycle of a Module or Service is to be initialized once, started once or many times until successful, and finally stopped. These should ideally be implemented to support accidental method calls in the wrong order (`init()` → `stopApp()`, `stopApp()`→ `init()`→ `startApp()` ...) and not trigger exceptions or return errors if this does not cause harm.

Some engineers may want to implement re-startable Module or Service. Whether one should call the `init(...)` method again before the `startApp()` method is called again, is up to the software engineer to decide. However, the implementation of a `restart(...)` method might be more appropriate.

About the Future

The progressive integration of OSGi in JXSE (level 2 and 3) will have an impact on the way modules and services are created and loaded on peers. This is work-in-progress. There are some fundamental limitations of service providers systems based directly on JAR files, since there is a level of abstraction missing in the Java Programming Language. This is well described by Neil Bartlett in the preview of his "*OSGi in Practice*" book[6].

The OSGi framework will help solving these issues. Hence, software engineers who need to implement services to load on peers would probably be better off waiting for the complete integration of OSGi if they can afford it. Else, they should strive at implementing OSGi services and bundle and load them via activators without referring to the JXTA/JXSE service and module interfaces for now.

Route Selection

The process of selecting a route is complex and performed by the `EndpointRouter` and the `RouteResolver`. Further work is expected on this topic in future releases, so we will only cover this globally:

- `EndpointRouter` – The endpoint router contains a `Destination` object which contains `Wisdom` objects for endpoint addresses of remote peers, that is, what we currently know about accessibility to those peers from either incoming or outgoing messages.

  `Destination` contains a `getCurrentMessenger(...)` method returning a messenger to the provided endpoint address if any is available.

  `EndpointRouter` contains an `ensureLocalRoute(...)` method which returns what `getCurrentMessenger(...)` returned if it is not `null`, else it calls the `findReachableEndpoint(...)` method.

  This method calls the `findBestReachableEndpoint(...)` method which does its best to collect endpoints to a remote peer, tests them, and prioritize them by giving priority to TCP connections and connections not involving relays. The result (if any) is returned.

Currently, JXTA/JXSE will only consider direct routes to target peers or routes involving only one hop.

6 See http://neilbartlett.name/blog/categories/osgibook/

Mavenization & Dependencies

Mavenization

Several members of the community have been requesting a Maven 'implementation' of the project. This has been achieved as part of the 2.6 release.

Directory Structures

Pom.xml Organization

Three Maven `pom.xml` files have created. A parent `pom.xml` is located in the main directory and two child `pom.xml` are stored in respectively `\jxse` and `\jxse-tutorial`.

Current Structure for Code

The JXTA/JXSE code is organized according to the following directory structure:

```
\RootDirectory
    \api           - JXTA/JXSE API
    \contrib       - 3rd-party contribution not included in core code
    \impl          - JXTA/JXSE implementation
    \jxse          - Maven directory for core JXSE code
    \jxse-tutorial - Maven directory for tutorial JXSE code
    \lib           - 3rd-party libraries
    \tests         - JXTA/JXSE JUnits
    \tutorials     - Tutorial code
```

This does not meet the standard maven directory structure requirements. Instead of re-organizing the code, which would have an impact on the current `Ant` build process, it has been decided to (temporarily) implement a pre-compilation step in the `pom.xml` file to copy code from the current structure to the maven directory structure.

A corresponding profile has been implemented for code copy and can be activated with:

```
mvn -P import-from-ant generate-sources
```

The code is imported respectively in the `\jxse` and `\jxse-tutorial` directories according to the standard Maven directory structure for JAR packaging. With this, Maven can be used in a normal fashion.

To return to the initial condition, one can issue a “clean” command as following:

```
mvn -P import-from-ant clean
```

Additions To The Build Cycle

Some additions have been made to the traditional Maven build cycle:

- Parent pom.xml:
  - `maven-compiler-plugin` – Source and target configuration have been set to Java 1.5.
- JXSE pom.xml:
  - Repositories – The http://repository.jboss.org/maven2 repository as been added for the netty library.
  - `maven-compiler-plugin` – Source and target configuration have been set to Java 1.5.

    - `process-classes` – The `maven-bundle-plugin` from Apache Felix has been added to add OSGi information in the MANIFEST.MF file in the `jxse.jar`. This is necessary to use JXSE as an OSGi bundle/service.
    - `test` – The `maven-surefire-plugin` has been configured to include and exclude JUnit tests. This is where software engineers can re-activate some existing tests if necessary.
    - `prepare-package` – The `maven-javadoc-plugin` has been configured to only generate Javadoc for API code.
    - `package` – The `maven-assembly-plugin` has been configured to generate a package artifact with the help of an assembly descriptor.
    - `package` – The `maven-antrun-plugin` has been added to generate artifacts that can be published on Kenai (tar, gz, zip, etc...).
    - `reporting` – The `maven-surefire-report-plugin` and `cobertura-maven-plugin` have been added in the reporting section.
    - `profile` – An `import-from-ant` profile with two phases (`generate-source` and `clean`) has been created to move code in a proper Maven directory structure.
    - `profile` – The `replace-if-jre-6` and `replace-if-jre-6` profiles have been included to tweak source code, in order to solve compilation issues when using Java 1.5 or 1.6.
- JXSE Tutorial pom.xml:
    - `profile` – The same `import-from-ant` profile with two phases (`generate-source` and `clean`) has been created to move code in the proper Maven directory structure.

Kenai Artifacts

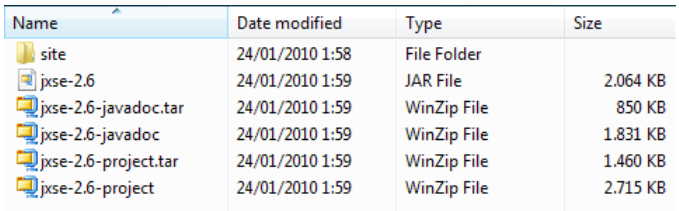
The Maven implementation is configured to generate artifacts to post on the jxse.kenai.com website. These artifacts are located in the `/jxse/target/kenai` directory and can be generated with the following set of commands (from the `/jxse` directory):

```
mvn clean

mvn site:site

mvn -Dmaven.test.skip=true package
```

The generated artifacts are:



Name	Date modified	Type	Size
site	24/01/2010 1:58	File Folder	
jxse-2.6	24/01/2010 1:59	JAR File	2.064 KB
jxse-2.6-javadoc.tar	24/01/2010 1:59	WinZip File	850 KB
jxse-2.6-javadoc	24/01/2010 1:59	WinZip File	1.831 KB
jxse-2.6-project.tar	24/01/2010 1:59	WinZip File	1.460 KB
jxse-2.6-project	24/01/2010 1:59	WinZip File	2.715 KB

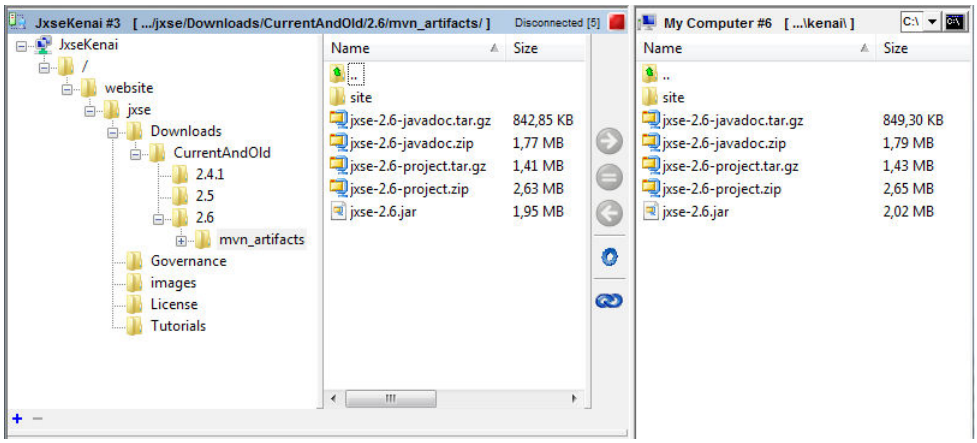
- The `/site` directory contains an `/apidoc` directory for online Javadocs.
- The `jxse-2.6-javadoc.*` files contains zipped version of the Javadoc.
- The `jxse-2.6-project.*` files contain the source code.
- `jxse-2.6.jar` is the compiled source code file.

These artifacts can be imported (as is) on the jxse.kenai.com website (using BitKinex[7] for example) in the:

```
/website/jxse/Downloads/CurrentAndOld/2.6/mvn_artifacts
```

7 More information available at: http://kenai.com/projects/help/pages/UploadWebsiteFiles

directory.



Dependencies

The delivered `jxse-2.6.jar` has the following dependencies, which should appear in the classpath:

Jar	Description
bouncycastle/bcprov-jdk15-145.jar	Mandatory: used for encryption
h2/h2-1.1.127.jar	Mandatory: used by the H2 implementation of the cache manager (Cm)
jetty/javax.servlet.jar	Mandatory: required by Jetty
jetty/org.mortbay.jetty.jar	Mandatory: required by the HTTP transport
netty/netty-3.1.5.GA.jar	Mandatory: required by the TCP transport
Optional runtime dependencies:	
derby/derby-10.5.1.1.jar	Optional: used by the derby based Cm implementation
felix/felix.jar	Optional: used to activate the OSGi framework
junit/junit-4.4.jar	Compile time: unit testing framework
jmock-2.5.1/*.jar	Compile time: mock objects are used in a number of tests

The mandatory Jars are required at runtime. At compile time, the mandatory and optional Jars must be available to perform the build process.

Under The Hood

The purpose of this chapter is to describe what is happening under the hood in JXSE code. One assumes that the reader is familiar with the Java programming language and with the JXTA specification document. The information provided here has been reversed engineered from the code and made available by former contributors to JXTA/JXSE.

The first part of this chapter will describe global code implementation concepts in JXSE. Then, we will cover the well-known JXTA services and specific features such as JXTA bidirectional pipes, JXTA sockets, etc...

Core Implementation Concepts

JXTA Loader

The JXTA loader abstract Java class (implemented via the RefJxtaLoader class) is the fundamental tool that JXSE uses to load Java byte code corresponding to modules or services used within an instance of the application.

This loader is based on a traditional URL Java class loader and offers a couple of extra methods to:

- Load or find the Class class corresponding to the implementation of module via its module specification ID.
- Define modules (i.e. Class Java classes) from their module implementation advertisement.
- Find module implementation advertisements from the corresponding module Class classes or module specification ID.

The `Class` classes are searched/loaded from either the `jxta.jar` file in order to start the application, or from the lib jars delivered with the application, or from any other URL location provided to the JXTA loader as a potential source of Java byte code. These URLs may point towards locations on the current device or anywhere on the Internet. This is very useful when one wants to distribute code in one location and have peers load that code remotely to offer services.

In the `RefJxtaLoader` class, a specific constant called MAX_XFER_TIME controls the maximum amount of time the loader will wait for content to load on the current peer when retrieved from a remote location. It is currently set to 60 seconds.

Just like peer groups, there are several instances of JXTA loaders (one per peer group) and these are organized according to the same hierarchy as the parent-child peer group hierarchy in the application. This is achieved via the `initFirst(...)` method of the GenericPeerGroup abstract class. An initial and root instance of the `RefJxtaLoader` class is created as a static attribute of the `GenericPeerGroup` abstract class when it is loaded. The parent loader of this JXTA loader is the Java class loader used to load the `GenericPeerGroup` abstract class itself.

Each time one tries to load a class via a JXTA loader, that JXTA loader will first ask its parent loader whether it can load the class (recursively to the root JXTA loader and above) and if not, will try to load it itself. Each instance of JXTA loaders has its own set of URLs. In the current JXSE model, parent loaders prevail on child loaders systematically.

Default Module Implementations

JXSE defines a set of default implementations for well-known modules and services. These are characterized by module specification IDs available from the `PeerGroup` interface in the `net.jxta.platform.Module` text file available in the META-INF.services package in the reference implementation source code. In other words, JXSE is using the JAR Service Provider concept.

The following table provides the list of Java classes corresponding to the default implementation of JXTA and JXSE services:

Module/Service	Java class implementation
`Resolver service`	`net.jxta.impl.resolver.ResolverServiceImpl`
`Discovery service`	`net.jxta.impl.discovery.DiscoveryServiceImpl`
`Pipe Service`	`net.jxta.impl.pipe.PipeServiceImpl`

None Membership Service	net.jxta.impl.membership.none.NoneMembershipService
Password Membership Service	net.jxta.impl.membership.passwd.PasswdMembershipService
PSE Membership Service	net.jxta.impl.membership.pse.PSEMembershipService
Rendezvous Service	net.jxta.impl.rendezvous.RendezVousServiceImpl
Peerinfo Service	net.jxta.impl.peer.PeerInfoServiceImpl
Endpoint service	net.jxta.impl.endpoint.EndpointServiceImpl
TCP Message Transport	net.jxta.impl.endpoint.tcp.TcpTransport
HTTP Message Transport	net.jxta.impl.endpoint.servlethttp.ServletHttpTransport
Router Message Transport	net.jxta.impl.endpoint.router.EndpointRouter
TLS Message Transport	net.jxta.impl.endpoint.tls.TlsTransport
JXME Proxy Service	net.jxta.impl.proxy.ProxyService
Relay Message Transport	net.jxta.impl.endpoint.relay.RelayTransport
Always Access Service	net.jxta.impl.access.always.AlwaysAccessService
Simple ACL Access Service	net.jxta.impl.access.simpleACL.SimpleACLAccessService
PSE Access Service	net.jxta.impl.access.pse.PSEAccessService
Cryptobased-ID Message Transport	net.jxta.impl.endpoint.cbjx.CbJxTransport
IP Multicast Message Transport	net.jxta.impl.endpoint.mcast.McastTransport
Content Service	net.jxta.impl.content.ContentServiceImpl

The content of the `net.jxta.platform.Module` text file is loaded by the default implementation of the JXTA loader to establish links between Java Class objects and well-known module specification IDs.

Each time the Java object instance of a service is required, the corresponding Java Class is extracted from that list and Java reflexion is used to create the object.

Peer Group Global Registry

The peer group global registry is a unique instance of a static class attached to the `PeerGroup` interface. Its purpose is to manage a list of instances of peer group classes together with their corresponding ID. The purpose of this registry is to make sure that one and only one instance of a peer group exists per ID within an instance of the application.

Typically, this registry used to make sure one and only one instance of the `WorldPeerGroup` is created within the application.

Peer Group Implementations

The `PeerGroup` interface is implemented by an abstract class called `GenericPeerGroup` in JXSE. This class contains the object instances corresponding to the implementation of the well-known JXTA services for the peergroup (Endpoint service, Resolver service, etc...) and of other services provided by the peer group. It also contains the peer's advertisement in this group, together with its module implementation advertisement and some configuration parameters.

A mechanism is also implemented that maintains the number of references to the peer group (for parent-child relationships for example), which uses three methods. It makes sure that a peer group is not stopped before all the references to it have been removed.

The `GenericPeerGroup` class is extended by several other classes directly or indirectly:

- *StdPeerGroup* – Extends the `GenericPeerGroup` class. It contains an instance of the `Cm` cache manager class. When this module is started, all existing SRDI index local entries for the peer group are deleted. Its services are started. The implementation advertisement of the group and of its discovery service are published locally using the DEFAULT_LIFETIME and default DEFAULT_EXPIRATION constants.
    - *Platform* – Extends the `StdPeerGroup` class. This class is the default implementation for the `WorldPeerGroup`. It contains an essential static method called `getDefaultModuleImplAdvertisement()` returning a module implementation advertisement containing the module class IDs and the module specification IDS of the standard/reference implementations of JXTA services/modules in JXSE. Contrary to other peer groups, the `WorldPeerGroup` does not have a parent group.
    - *ShadowPeerGroup* – Extends the `StdPeerGroup` class. This class is the default implementation for the NetPeerGroup and retrieves its configuration parameters from its parent peer group (contrary to the `Platform` peer group).

Other peer groups are created by software engineers by calling one of the `newGroup()` methods of existing peer group instances. They may or may not provide module implementation advertisements for the creation of new groups. If not, the one from the parent group (i.e. on which the `newGroup()` method is called) will be used.

Starting the Network

Before one starts the JXTA Network, an instance of the `NetworkManager` is created. A couple of parameters such as the connection mode (ADHOC, EDGE, RENDEZVOUS, etc...) and eventually a location to persist configuration parameters are provided.

When the JXTA network is started, the following happens:

1. If the JXTA network has already been started, nothing happens. The `NetPeerGroup` is already available.
2. Else, if no configuration is available, one is create based on the specified connection mode.
3. Then, an instance new of the `NetPeerGroup` factory class is created.
4. The NetPeerGroup factory creates a `WorldPeerGroup` factory.
5. The `WorldPeerGroup` factory creates a new `WorldPeerGroup` instance by fetching the name of the class corresponding to its default implementation in JXSE, that is: Platform. This is performed with the help of the JXTA loader.
6. The services offered by the `WorldPeerGroup` are loaded and started.
7. The implementation advertisements of the `WorldPeerGroup` and of its discovery service are published locally.
8. A new instance of the `NetPeerGroup` is created (using the `ShadowPeerGroup` class) and returned by the `NetPeerGroup` Factory. `WorldPeerGroup` is set as the parent peer group.

   The services of the `NetPeerGroup` object will have been started too.

SRDI Manager, Index & Cache Manager

JXSE uses two objects to manage SRDI indexes: `SrdiManager` and `Srdi` (since release 2.6, the `SRDI` class has been renamed into `SrdiManager` and the `SrdiIndex` class has been renamed into `Srdi`). These keep track of advertisement keys (not the whole advertisement) on the local peer, but also keys of advertisements located on other peers, depending on their role. These two objects work in combination with a third object, the `CacheManager` storing the advertisements themselves.

Typically, an EDGE peer will only exchange SRDI information with the RENDEZVOUS it is connected to. RENDEZVOUS exchange random extracts of SRDI information they possess with other RENDEZVOUS they are connected to. These are called *replica* peers.

The `SrdiManager` class implements the `RendezvousListener` interface to process messages exchanged between rendezvous or peers. It also creates a `SrdiManagerPeriodicPushTask` object which pushes

SRDI entries at a regular rate. `Srdi` objects are created by peergroup services: discovery service, the `RouteResolver` class and the `PipeResolver` class which we will cover later.

- `DiscoveryService` – When this service is started, its behavior depends of the connection mode.

  If it is a rendezvous, it will not track deltas (i.e. modifications in the local manager). A SRDI is created together with a corresponding SRDI manager. The discovery service registers as a SRDI messages handler in the resolver service.

  If we are an edge and if we are connected to a rendezvous, deltas are tracked. No `Srdi` object is created, but a `SrdiManager` is. It is a attached `SrdiManagerPeriodicPushTask` which is started to push delta entries keys to the connected rendezvous.
- `RouteResolver` – The route resolver maintains route advertisement SRDI entries in a `Srdi` object. A corresponding `SrdiManager` is created too. However, the router resolver does not push its entries automatically. It is the corresponding `EndpointRouter` that does via the `setRoute()` method when new routes are available.
- `PipeResolver` – The pipe resolver create a `SRDI` class to store input pipe information. If this peergroup is an edge deltas are pushed to the connected rendezvous. If it is a rendezvous, entries are replicates to replica peers.

SRDI Manager

The SrdiManager class offers several methods:

- `replicateEntries()` - This method takes a SRDI message in parameter, extracts its entries, selects a random peer from the local peer view (called a replica peer) for each entry and pushes a new SRDI message based on the primary key contained in the parameter SRDI message to the selected peer using the pushSrdi() method.
- `pushSrdi()` - This method creates a resolver SRDI message, invokes the resolver service of the peergroup and calls the `sendSrdi()` method.
- `forwardSrdi()` - These methods forward a query using the resolver service of the peergroup.
- `forwardSrdiMessage()` - This method builds a Srdi message from parameter elements and sends it to a target peer.
- `RendezvousEvent()` - This method captures Rendezvous events. If a rendezvous disconnects or a rendezvous connection failed, the local SRDI entries are republished to other peers.

REM: A Srdi manager object will only replicate entries if its peerview contains at least 2 entries (cf. RPV_REPLICATION_THRESHOLD).

SRDI

The Srdi object implements the `runnable` interface. This is the object creating SRDI entries on the local peer. Several methods are available to manipulate entries: `add()`, `getRecord()`, `getIndexMap()`, `clear()`, etc... Entries are published with a lifetime. The `run()` method periodically calls the `garbageCollect()` method to remove expired entries. The `stop()` method stops the garbage collection thread and closes all opened files.

Cache Manager

Peers need to store advertisements locally on peers to operate on the JXTA network. This is performed with the help of the `CacheManager` class. `CacheManager` object instances are created by peergroup object instances when these are initialized.

Similarly to the Srdi class, the `CacheManager` object creates files entries, unless the in-memory implementation is selected. The object offers several methods to manage records and implements a garbage collection system too. The cache manager 'tracks deltas', that is, changes in the local published advertisements.

Well-known Services

The following pages describe what is happening in the implementation of the JXTA services in JXSE. The author is assuming that the reader is familiar with these services. One should keep in mind that services are subtypes of modules and that every instance of a service is a Java object attached to a peergroup.

With a twist...[8]

The service object instances returned by the `GenericPeerGroup` class are not direct instances of the Class objects defined by the JAR Service provider, but rather interfaces to them. `GenericPeerGroup` calls the `getInterface()` method on the instance of the well-known service it possesses.

For example, the `GenericPeerGroup.getEndpointService()` does not return an instance of `EndpointServiceImpl`, but an instance of `EndpointServiceInterface`, since a call to `EndpointServiceImpl.getInterface()` will return such an object.

The returned 'interface' also implements the `EndpointService` interface, together with corresponding methods, in a slightly different way than the `EndpointServiceImpl` object. It also the calls methods on the 'real thing' `EndpointServiceImpl` object it knows via its constructor.

The consequence is that software engineers must not only take a look at Java class implementation of the well-known services in the `net.jxta.platform.Module` text file of the META-INF.services package to understand how JXSE implements JXTA. They should first take a look at the corresponding 'interface' objects returned by the getInterface() methods of these classes, since all implementations of peergroups in JXSE rely on the `GenericPeerGroup` class.

Before we start describing the implementation of the endpoint service (and subsequently other well-known services), we will first describe several implementation concepts of the transportation layer.

Transportation Layer

The following pages describe fundamental concept of the transporation layer implementation in JXSE.

Message Transports

The message transports of a peer are responsible for sending and receiving message to and from other peers over the JXTA network, using whatever protocol they wish. There are three types of message transports: message receivers, message senders and messages propagators (for propagated messages like multicasting) in JXSE.

Several classes implement the message sender, receiver and propagator protocols:

Implementing Classes	Message Sender	Message Receiver	Message Propagator
`TcpTransport`	X	X	
`McastTransport`			X
`HttpMessageReceiver`		X	
`HttpMessageSender`	X		
`TlsTransport`	X	X	
`RelayClient`		X	
`RelayServer`	X		
`EndpointRouter`	X	X	
`CbJxTransport`	X	X	

- `TcpTransport` - This class implements the TCP Message Transport.
- `McastTransport` - This class implements the TCP Message Transport.
- `HttpMessageReceiver` - Simple Message Receiver for server side..
- `HttpMessageSender` - This class implements the TCP Message Transport.
- `TlsTransport` - Implementation which uses TLS sockets..
- `RelayClient` - RelayClient manages the relationship with the RelayServer(s).
- `RelayServer` - Relay server that maintains outgoing message queues, leases, etc.
- `EndpointRouter` - This class implements the TCP Message Transport.

8 This sub-section is now obsolete, since this service interface mechanism has been disactivated in the code. It was left in this document for those who want to understand earlier versions of the code.

- `CbJxTransport` - Message transport implementation which which provides message verification by examining message signatures (this has not been activated in the code yet).

Message transports are registered in each endpoint service object instances according to type of communication layer which is enabled (or not) by the configuration.

Typically, messengers are retrieved from message transports in order to send messages. For example, calling `getMessenger()` on TcpTransport will return either a `LoopbackMessenger` if the destination is one of the public addresses of the transport or a new TcpMessenger otherwise. We will cover the `Messenger` interface and this topic further.

Message Sender

The `MessageSender` interface extends the `MessageTransport` interface. It offers a `getMessenger()` method which returns a `Messenger` for a given Endpoint address. This messenger can later be used to send messages.

Each message sender contains the endpoint address where replies to sent messages should be sent. A `MessageSender` can be connection oriented (like TCP/IP), meaning that series of messages can be transported efficiently to the destination. It can also allow routing, meaning that it can be used as a intermediary step (hop) to forward messages to peers who are not directly reachable via this transport.

Message Receiver

The `MessageReceiver` interface extends the `MessageTransport` interface. It receives messages from the network. It also defines one method called `getPublicAddresses()` which returns the list of public address from which it is reachable.

Message Propagator

The MessagePropagator interface extends the MessageTransport interface. It defines a `getPublicAddress()` method which returns the endpoint address of the sender, together with a `propagate()` message to send/propagate messages to multiple peers.

Messengers

Messengers are implemented to send messages to a destination. A messenger contains methods to retrieve the endpoint destination address and the maximum message size it is capable of handling. Wherever applicable, this interface also defines a `getChannelMessenger()` method to return another messenger to the same destination if available. We will cover channel messengers later.

Several sending methods are defined in the Messenger interface. Some will throw an `IOException` is the message is malformed or if the Messenger is not usable anymore. Some will invoke an `OutgoingMessageEventListener` when provided.

Software engineers can also monitor the status of a sent message by calling the `getMessageProperty(Messenger.class)` method on it. This will return an `OutgoingMessageEvent` object which can be compared to `OutgoingMessageEvent.SUCCESS` or `OutgoingMessageEvent.OVERFLOW` (REM: It is not because a message has been successfully sent from a peer that guarantees that it will necessarily reach its destination).

Several statuses are defined for messengers:

Status	Description
UNRESOLVED	No message was ever submitted for sending. No connection has ever been attempted.
RESOLVING	Initial connection is being attempted. No message is pending.
CONNECTED	Currently connected. No message is pending (being sent implies pending).
DISCONNECTED	Currently not connected. No message is pending.
RESOLPENDING	Initial connection is being attempted. Messages are pending.

Status	Description
RESOLSATURATED	Initial connection is being attempted. Messages are pending. New messages may not be submitted at this time.
SENDING	Currently connected and sending messages.
SENDINGSATURATED	Currently sending messages. New messages may not be submitted at this time.
RECONNECTING	Currently trying to re-establish connection. Messages are pending.
RECONSATURATED	Currently trying to re-establish connection. New messages may not be submitted at this time.
RESOLCLOSING	Attempting initial connection. Close has been requested. Messages are pending. New messages may no longer be submitted.
CLOSING	Currently sending messages. Close has been requested. New messages may no longer be submitted.
RECONCLOSING	Trying to re-establish connection. Close has been requested. Messages are pending. New messages may no longer be submitted.
UNRESOLVING	Failed to establish initial connection. Pending messages are being rejected. New messages may no longer be submitted.
BREAKING	Failed to re-establish connection. Pending messages are being rejected. New messages may no longer be submitted.
DISCONNECTING	Breaking established connection for expedite closure. Pending messages are being rejected. New messages may no longer be submitted.
UNRESOLVABLE	Failed to establish initial connection. New messages may no longer be submitted. State will never change again.
BROKEN	Failed to re-establish connection. New messages may no longer be submitted. State will never change again.
CLOSED	Closed as requested. All pending messages could be sent. New messages may no longer be submitted. State will never change again.

Messenger Implementations

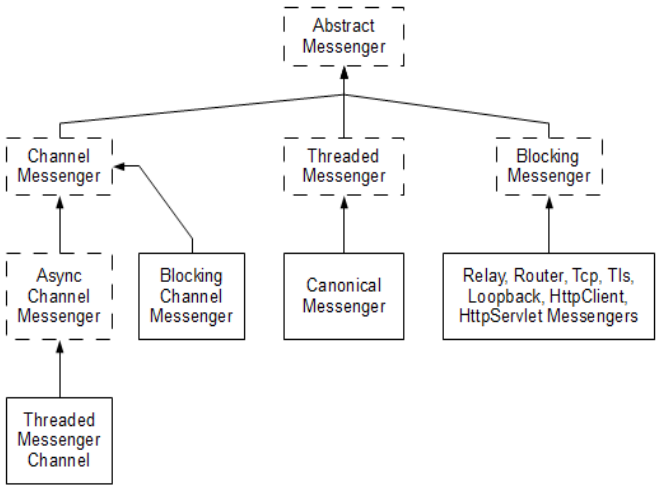
Class (with inheritance level)	Description
AbstractMessenger	Base abstract class for implementation of all messengers.
- ChannelMessenger	Abstract class messenger for sub-destinations of a destination, like a service for example.
- - AsyncChannelMessenger	Abstract class messenger to provide asynchronous message sending via queuing.
- - - ThreadedMessengerChannel	Private class of the ThreadedMessenger returned when its getChannelMessenger(...) method is called.
- - BlockingMessengerChannel	Private class of the BlockingMessenger returned when its getChannelMessenger(...) method is called.
- BlockingMessenger	Abstract class for 'real' transportation messengers.
- - RelayMessenger	Protected static class of RelayServerClient returned when getMessenger(...) is called.

Class (with inheritance level)	Description
- - RouterMessenger	Class implementation for destinations which are logical peers.
- - HttpClientMessenger	Simple messenger that simply posts a message to a URL.
- - HttpServletMessenger	Simple messenger that waits for a message to give back to the requesting client
- - TcpMessenger	Implements a messenger which sends messages via raw TCP sockets
- - TlsMessenger	This class implements sending messages through a TLS connection
- - LoopbackMessenger	Messenger delivering local messages when the source peer is identical to the destination peer.
- ThreadedMessenger	Abstract class shared by multiple channels to automatically distribute the available bandwidth among the channels.
- - CanonicalMessenger	Private classes of the Endpoint service implementation returned when the getCanonicalMessenger(...) method is invoked.

The table above describes the classes implementing the `Messenger` interface. There are three types of messengers serving different purposes; blocking messengers, canonical messengers and channel messengers:

- `BlockingMessenger` – These messengers are physically sending messages to others peers, with the exception of the `RouterMessenger` which searches for routes to other peers (if necessary) before sending them via other blocking messengers.
- `CanonicalMessenger` – These are non-blocking messengers storing messages to a destination peer in a queue. They are linked to at most one and only one blocking messenger at any given time to transmit messages to other peers. They remain open as long as the destination is reachable, even if the underlying messenger may change.
- `ChannelMessenger` – These messengers are used to send message to a specific destination on a destination peer (for example the rendezvous service, the pipe service, etc...). They are attached to the canonical messenger for the destination peer.

Most often, channel messengers are used by the application as intermediaries to send messages, but sometimes, they are skipped when a direct messenger is used. Typically, a direct messenger is a messenger capable of direct connection and communication to another peer (for example, through TCP).



The above diagram show the inheritance structure of messenger classes.

Endpoint Service

The `EndpointServiceImpl` class implements the JXTA discovery service via the `EndpointService` and the `MessengerEventListener` interface. Each instance its own collection of MessageTransports. Each transport registers itself in the Endpoint service when created.

Using Message Transports

- The Endpoint service implements[9] the `getMessageTransport()` method which is typically called with the "jxta" or "tcp" parameter by the well-known services. Using the "jxta" parameter will return the instance of the EndpointRouter sending messenger, upon which the `getMessenger()` method can be called.

Depending on the provided parameters, the returned object can be a `LoopBackmessenger` if the target peer is this peer or a RouterMessenger:

- `LoopBackmessenger` – Each time a message is sent with this messenger, it creates a runnable which will invoke the `processIncomingMessage()` of the current endpoint service.
- `RouterMessenger` – These messenger are for virtual peer destinations. A virtual peer is a JXTA address which is based upon a peer ID rather than on a physical network address. When created, the RouterMessenger will invoke the `getGatewayAddress()` method from its EndpointRouter, which will convert the virtual address:
  - Either into a direct non-virtual endpoint address if the peer can be reached directly
  - Or into the address of the first peer along a route leading to the destination peer

    REM: If no route is available, the peer will use the routing protocol to try to fetch one route from other peers in the peergroup for 30 seconds.
  - If the conversion is unsuccessful, the creation of the RouterMessenger is aborted by throwing an `IOException`.

  When a message is sent, a Messenger for the converted address is retrieved and used to send the message. This messenger is typically directly 'connected/resolved' to the destination peer.

Well-known services can also call the `getMessageTransport()` method to retrieve all available message transports on the endpoint service and loop them to retrieve a Messenger.

Retrieving Messengers

The 'interface' version of the endpoint service (cf. `EndpointServiceInterface`) implements the `getMessenger()` method(s) in a different way than the default implementation[10]:

1. `getMessenger()` will try to retrieve an immediate messenger through the `getMessengerImmediate()` method:
   - This method will check whether a channel messenger already exists for the provided endpoint address and return it if it is in a USABLE state.
2. Else, the EndpointServiceInterface calls the `getCanonicalMessenger()` method of the EndpointServiceImpl, which loops through its map of messenger to check whether a usable one is available. Any unusable canonical messenger is deleted.
   - If none is available, we check whether a message sender for the endpoint address type (TCP, HTTP, virtual peer address...) is available within the endpoint service. If yes, a new canonical messenger instance is created and returned.
   - Else, if a parent peergroup is available, we try to retrieve the a canonical messenger from it.
   - Otherwise, `null` is returned.
3. If no canonical messenger can be found (i.e. the endpoint address type is not supported by the endpoint service), `getMessengerImmediate()` returns null.

9 The 'interface' version of the Endpoint service implementation is neutral to message transports. It directly calls the 'real thing' object methods.

10 'Interface' objects have been disactivated and the code they contained has been moved to the real implementation of service interfaces.

- Else, we retrieve a channel messenger from the canonical messenger. We double-check that no channel messenger has been created since we first checked. If yes, we return it. Else, we register the new channel messenger and return it.

4. If no channel messenger is returned, the `getMessenger()` returns null too.
5. Else, we explicity request the messenger to resolve (i.e., to try to 'connect' to the target peer) within one minute.
6. If successful, we return the messenger, else we return `null`.

The returned messenger (if available) can be used to send messages. Several features retrieve messengers from the endpoint service (for example: JxtaSocket, JxtaBidiPipe, output pipes...).

Incoming Messages

When a message is arriving at an instance of an endpoint service, the following happens:

1. If the source peer (i.e., sender) is `this` peer, then the message is a loop-back. It is discarded.
2. If the source address or the destination is `null`, the message is discarded too.
3. If the destination service name is `null` or of length 0, the message is discarded too.
4. The message is passed through any existing filter and may be discarted (i.e., filtered out).
5. We try to retrieve a listener for the corresponding service name and parameter.
6. If none is available, the message is discarded.
7. Else, the listener is called.

Diagrams

The endpoint service is complex. Some diagrams have been provided in Appendix A to facilitate the understanding of this service.

Messenger Methods Callers

The following table indicates which parts of the code calls which messenger method at the `EndpointService` level:

Method	Caller
`GetMessenger()`	`JxtaSocket via LightweightOuputPipes`
	`JxtaBibiPipe via LightweightOuputPipes`
	`PeerConnection – To remote peers`
	`AdhocRdvService – For message propagation`
	`BlockingWireOutputPipe – To get usable messengers`
	`NonBlockingOutputPipe – For asynchronous sending`
	`RelayServer – For lease requests`
`GetMessengerImmediate()`	`Destinations – To get wisdom about routes`
	`ResolverService – To send messages`
	`RendezvousService – To propagate messages, send disconnects, monitor connections with remote peers...`
	`PeerView & PeerView Elements – To send peerview messages`

Resolver Service

The `ResolverServiceImpl` class implements the JXTA resolver service using the `ResolverService` interface[11]. Amongst other things, this object contains a list of query handlers and SRDI handlers, together with a reference to the peergroup's Endpoint service and Membership service.

During the initialization of the service, a unique ID string is created as a base to create unique handler names. Then, an (incoming) query listener, a query response listener and a SRDI listener are created with respectively the `DemuxQuery`, `DemuxResponse` and `DemuxSrdi` private classes. These listeners are registered in the Endpoint service.

- Typically, `DemuxQuery` implements the `EndpointListener` interface. Each time a message arrives, the 'query' part is extracted and processed. If a matching query handler exists (i.e., has been registered in the resolver service), the query is transmitted to it.

  If no query handler can be found and if the current peer is acting as a Rendezvous for its peer group, the query is propagated to other known rendezvous who may have a handler for it (or who may forward it themselves). A loop and TTL (time to live) control system is implemented to check sent messages. The query is also forwarded to all the peers of the peergroup reachable on the local network (using multicasting or broadcasting).

  If no query handler can be found and if the current peer is not acting as a Rendezvous for its peer group, the query is propagated to all other peers using all available message transports.
- The `DemuxResponse` class implements the EndpointListener interface too. When a message arrives, the 'query response' part is extracted and processed.

  If a matching query handler exists (i.e., has been registered in the resolver service), the query response is transmitted to it. Else, the message is ignored.
- The `DemuxSrdi` class also implements the EndpointListener interface. When the message arrives, its content is extracted and processed.

  If a corresponding Srdi handler is available, the message content is transmitted to it. Else, it is ignored.

When a query is sent to a destination peer, a route back to the sending peer may be included if this is not available in the message. This may facilitate the work of the destination peer to send back answers, since often, it may not possess that route information.

If a destination peer is specified, a messenger to the destination is retrieved from the Endpoint service and the message is sent.

If no destination peer is specified and the current peer is acting as a Rendezvous for its peer group, the query is sent to other known rendezvous. The query is also forwarded to all the peers of the peergroup reachable on the local network (using multicasting or broadcasting). If the current peer is not acting as a Rendezvous, the query is sent to all other peers using all available message transports.

A similar process is used to send query responses. If a route to the issuing peer is available, it will be used as a hint to send the response back.

All Srdi message sent by the resolver service are zipped before sending.

When the resolver service is stopped, all registered listeners in the Endpoint service are removed.

Discovery Service

The `DiscoveryServiceImpl` class implements the JXTA discovery service via the `DiscoveryService` interface (amongst others). This object contains a list of discovery listeners, together with a reference to the peergroup's Resolver service, Rendezvous service and Membership service. It also contains a local instance of a `srdi` and a `srdiIndex` object.

During initialization, the `DiscoveryServiceImpl` object registers itself as a query listener in the resolver service and in the rendezvous service.

When advertisements are retrieved remotely:

- A unique query ID is created based on an Java atomic integer.
- Then, a discovery query is created using provided parameters.

11 The 'interface' version of the resolver service implementation is completely neutral; it always forwards method calls to the 'real thing' implementation object as is.

- If a discovery listener is provided, it is registered within the discovery service together with the query ID.
- A resolver query message is created and the discovery query is 'embedded' into it, together with the current peer ID (i.e., source/sender).
- If a target peer is specified or no SRDI index is available, the query is forwarded using the `sendQuery()` method of the resolver service.
- Else the discovery service will do its best to select proper target peer IDs from the SRDI entries, using attribute and value parameters when available. If such entries are available the `forwardQuery()` method of the Srdi object will be used. Else, the discovery service will revert to the `sendQuery()` method of the resolver service. It is not because because we do not know proper target peers that other peers don't.

When queries are received by target peers who can answer them, the `respond(...)` method is called, which crafts a query response and sends it to the source peer. The private `rawSearch()` method can be used to search for local advertisements (i.e., in the cache manager) to answer queries.

The discovery service implements other methods to publish advertisements locally or remotely, together with publication method. In the latter case, remote publications are performed using the same mechanisms and methods used to answer queries.

Processing SRDI Messages

The discovery service implements the `SrdiHandler` interface and processes SRDI messages via a couple of methods:

- `processSrdi()` - This method takes in a `ResolverSrdiMsg`, extracts its SRDI entries, add them to the SRDI index and replicates the entries to other RDVs.
- `pushSrdi()` - This method is used to push entries from this peer to other peers and ensure replication of srdi entries.

The discovery service registers itself as a `SrdiHandler` in the resolver service when the peer is a RENDEZVOUS or becomes one.

Switching between RENDEZVOUS/EDGE

The discovery service uses two methods to switch its behavior between EDGE and RENDEZVOUS:

- `beEdge()` - if a `RENDEZVOUS`, track deltas in the cache manager only and unregister this discovery service as a `SrdiHandler`. Recreates a `Srdi` instance.
- `beRendezVous()` - Switches the RDV boolean flag and stops tracking deltas. Recreates a `Srdi` instance and registers as a `SrdiHandler` in the resolver service.

Rendezvous Service

RendezVous Service Implementation

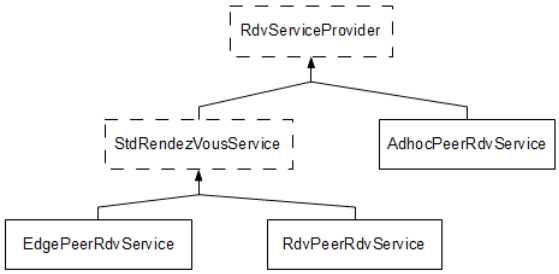
The `RendezVousServiceImpl` final class implements the JXTA rendezvous service via the `RendezVousService` interface[12]. This class contains (amongst others) references to the peergroup's endpoint service and a list of rendezvous listeners. When initialized, information such as the auto-rendezvous status and check interval is retrieved from the configuration.

If this peergroup is the `WorldPeergroup`, then the rendezvous service adopts an AD_HOC rendezvous configuration (which we will describe later). Else, according to the connection mode of the peer (AD_HOC, EDGE or RENDEZ_VOUS), an instance of the `AdhocPeerRdvService` class, the `EdgePeerRdvService` class or the `RdvPeerRdvService` class is created. This object instance is referred to as the *provider*. In other words, the rendezvous service behaves in different ways according to the connection mode.

All providers extend the `RendezVousServiceProvider` abstract class (directly or indirectly), which implements the `EndpointListener` interface. When started, it registers itself in the endpoint service of the

12 The 'interface' version of the rendezvous service implementation is completely neutral; it always forwards method calls to the 'real thing' implementation object as is.

peergroup for rendezvous propagate messages and unregisters when it is stopped. When it receives a message , a check on the TTL (time to live) and on possible loop (by checking whether this peer ID is already in the propagation list) is made to discard it if needed. Else, if we have a local listener for the message, it is transferred to it.



The `RendezVousServiceProvider` class also provides abstract methods to challenge rendezvous (i.e., check that leases are still alive within a given delay), to connect to other rendezvous, to disconnect from rendezvous, to retrieved the list of connected peers, to propagate messages, to indicate whether the peer is connected to a rendezvous, to walk messages to other rendezvous peers, to repropagate messages and to propagate messages in the peergroup. It also defines a set of methods to manage propagation headers in rendezvous propagate messages (add, update, remove, extract, etc...).

A `RdvWatchdogTask` private class is also defined to manage the auto-rendezvous status. If it is set and if the current peergroup is not the `WorldPeerGroup`, a periodic task is created to check whether the peer should become a rendezvous for its peergroup:

- If the peer is not a rendezvous and if it did not manage to connect to a rendezvous, it becomes one.
- If the peer is a rendezvous, we check whether we can demote ourselves:
  - If the peer view contains more rendezvous than the DEMOTION_MIN_PEERVIEW_COUNT limit (currently hardcoded at 5) and if no peers are connected to this rendezvous, it stops behaving as a rendezvous.
  - Else if there are less than DEMOTION_MIN_CLIENT_COUNT (currently hard-coded at 3) clients connected to the rendezvous, there is 1 chance out of 20 that it will stop behaving as a rendezvous (cf. the DEMOTION_FACTOR set at 0.05).

If a demotion or a promotion happens, the Rendezvous service changes of `RendezVousServiceProvider` instance, by creating a new ones.

Standard RendezVous Service

The stdRendezVousService abstract class extends the `RendezVousServiceProvider` class. It defines a timer to schedule tasks. When started, it registers a StdRdvProtocolListener in the endpoint service and unregisters it when stopped.

This class implements some of the abstract method defined in the `RendezVousServiceProvider` class. It also defines some abstract methods to retrieve peer connections (described next) and offers a method to send a message to each connection. It also provides a method to send disconnect requests to remote peers.

When a message from another peer arrives, this class checks whether with have a connection with it and if not, it sends a 'disconnect' to it (unless we are not an edge peer ourselves). In any case, the message is transmitted to `RendezVousServiceProvider` super class too.

PeerConnection

The `PeerConnection` is an abstract class implementing the `OutgoingMessageEventListener` and is used to manage leases between RENDEZVOUS and EDGEs, and connections between rendezvous via the `RdvConnection` and `ClientConnection` classes. It maintains a boolean indicating whether one believes that we are still connected to the remote peer or not. It also keep tracks of the amount of milliseconds we believe the connection will timeout unless when it is renewed.

The `PeerConnection` class also implements a `getCachedMessenger()` method to retrieve a `Messenger` for communication to the remote peer. A `sendMessage()` is method is also implemented to

send messages with the `Messenger`. If a message fails to be sent, the connection status will be set to false (unless this is caused by a queue overflow).

RendezVous Service Providers

For each type of provider:

- `AdhocPeerRdvService` – Extends the `RendezVousServiceProvider`. Ad hoc peers do not connect to rendezvous peers, but their rendezvous service will nevertheless forward and propagate rendezvous messages when received from other peers.
- `EdgePeerRdvService` – The edge peer rendezvous service extends the `StdRendezVousService` (which extends the `RendezVousServiceProvider` abstract class) and manages connections to rendezvous from an EDGE perspective. The object also contains an instance of a `SeedingManager` and a list of current connections with rendezvous.

  An edge peer tries to connect to one and only one rendezvous (cf. MAX_RDV_CONNECTIONS) and will check its connection every 15 seconds (cf. MONITOR_INTERVAL). The lease margin is set to 5 minutes and the TTL is set to 200 by default. These can be modified through configuration.

  A `MonitorTask` private class is implemented to managed connections to rendezvous. It tries to renew an existing lease before expiration if necessary and tries to create new ones using seeds when necessary.
- `RdvPeerRdvService` – The rendezvous peer rendezvous service also extends the `StdRendezVousService`, and manages leases with EDGE and RENDEZVOUS peers. When started, existing configuration is retrieved or default values are used if not available. The lease margin is set to 5 minutes and the TTL is set to 200 by default. The default maximum number of connections with clients is set to 200 (cf. DEFAULT_MAX_CLIENTS).

  A `StdRdvRdvProtocolListener` private class handles lease requests and disconnect requests from edge peers. The `RdvPeerRdvService` also refersto a `ClientConnection` class in order to handle connections with other peers and to a PeerView class which listens for peerview messages from the Endpoint address.

  When the service is started it creates a garbage collection recurrent task checking the status of 'client' connections and remove leases when necessary.

Pipe Service

The `PipeServiceImpl` class implements the JXTA pipe service via the `PipeService` interface. During initialization, it creates a `PipeResolver` class for the resolution of unicast pipes and a `WirePipeImpl` class for the resolution of propagate pipes.

The `PipeResolver` class maintains a map of (pipe ID, input pipe) pairs and a map of (pipe ID, set of output pipe listeners) pairs. When instantiated for a given peer group, the object registers as a listener in the corresponding resolver service. New instances of the `Srdi` and `SrdiIndex` objects are created. The `PipeResolver` instance also registers as a SRDI handler in the resolver service.

When a query containing a pipe resolver message arrives and the destination is `any` peer or `this` peer, the pipe sends a response mentioning whether a matching input pipe is available or not (i.e., whether the resolution is successful). Else, if the peer is a RDV, the SRDI objects are invoked to forward the message to other appropriate peers (maximum 20). If such peers IDs are not available from the SRDI or if the peer is an edge, we send a response to the issuer using the resolver service.

When a pipe resolver message response arrives, the responding peers advertisement is extracted and published locally using default expiration and default lifetime. If the response is negative, any matching entry in the local SRDI is removed, else it is created or renewed with an expiration of 10 minutes. The output pipe listener (if any is available) is notified either with a resolution event or a NACK (no acknowledgment) event.

The `PipeResolver` class is capable of receiving SRDI messages, and therefore processes the content (i.e., save it in the local SRDI) to facilitate the resolution of pipes. When such messages are received, they are propagated to other peers unless the information is related to propagated peers.

The `WirePipeImpl` class is implemented for propagated pipes at a peergroup level. It implements the `EndpointListener` interface and registers itself in the endpoint service of the peergroup. It also maintains a set of unique `WirePipe` class instances per pipe ID when `InputPipeImpl` objects are created through it.

When messages arrive, the `WirePipe` instance corresponding to the Pipe ID is retrieved, and if it exists, the message is transmitted to it. When a `WirePipeImpl` object is stopped, existing wire pipes are closed one by one and the object removes itself from endpoint incoming listeners' list.

Contrary to the `PipeResolver` class, the `WirePipeImpl` class does not implement SRDI objects.

When an input pipe is created via a call to `createInputPipe()` in the `PipeServiceImpl` class, one of the following object is returned according to the pipe type:

- `UnicastType` – An `InputPipeImpl` is created by providing the PipeResolver class instance. If a pipe message listener is provided, a synchronized queue of size 100 is created within the `InputPipeImpl` object. Then, the new `InputPipeImpl` is registered in the PipeResolver.
- `UnicastSecureType` – A `SecureInputPipeImpl` (which extends `InputPipeImpl`) class instance is created. The constructor calls the super constructor.
- `PropagateType` – An `InputPipeImpl` is created using the `WirePipeImpl` class.

When an output pipe is created via a call to `createOutputPipe()` in the `PipeServiceImpl` class:

If the number of peers who can (potentially) resolve the pipe is 1, a new output pipe is created via the `BlockingWireOutputPipe` object. Its constructor calls an essential private method called `checkMessenger()`. This method checks whether a Messenger to the remote peer is available and whether it is usable. If not, it tries to retrieve a direct one from the endpoint service or a regular messenger if it is not successful.

Else, the output pipe is created by calling the `wirePipe.createOutputPipe()` method which returns a `NonBlockingWireOutputPipe`. This pipe can handle multiple destination peers.

When sending a message with the `BlockingWireOutputPipe` object, one checks (twice if necessary) whether a messenger is available and tries to send it using the `sendMessageB()` method. If unsuccessful, an `IOException` is thrown.

When sending a message with the `NonBlockingWireOutputPipe` object, the `wirePipe.sendMessage()` method is called:

- `PropagateType` – If this peer belongs to the list of target peers, the local input pipe listeners are called. If the list of peers is empty (i.e. the message is to be sent to every peer) and if the peer is a rendezvous, or if the list of target peer is not empty, the message is propagated via the rendezvous service. Else, it is walked through the rendezvous service.
- `UnicastType` and `UnicastSecureType` – An output pipe listener is created and registered in the corresponding list. An input pipe resolution query is sent to the resolvable peer (or any peer if no peer is specified). We also check for local listener.

When the service is stopped, the `WirePipeImpl` class and the `PipeResolver` class are stopped too. All listener lists are cleared.

Simple Pipe Communication

Simple pipe communication implies the creation of an output pipe on an output peer and the creation of an input on an input peer. For the sake of simplicity, we will consider that both peers are different, although communication could happen on the same peer. As a reminder, once an output pipe is created, it tries to resolve to an input pipe.

PipeService – Create Input Pipe

When an input pipe is created (regardless of its type), an input pipe listener may be provided that is notified of new messages. The PipeService creates the proper object InputPipe which registers itself in the PipeService's PipeResolver object. The resolver handles pipe resolution queries coming from output pipe peers and will answer those addressed specifically to the registered input pipe. The resolver also registers the input pipe as an incoming message listener in the endpoint service.

Later, when a message arrives for the input pipe peer, these are either stored on a queue for later polling or directly sent to the corresponding listener if such is available. REM: if the queue is full, some messages may be lost/discarded without notice.

PipeService – Create Ouput Pipe

When an output pipe is created and depending on the selected creation method, the pipe service will attempt the resolution with an input pipe itself before returning the output pipe object, or it will 'remember' the provided output pipe listener and send a pipe resolution query.

OutputPipe Send

The output pipe implementation for simple pipe communication is the `BlockingWireOuputPipe` object. When sending a message:

1. The destination messenger is checked. If it is available and USABLE, the check stops successfully here.
2. Else, we try to retrieve a directed messenger from the endpoint service. If none is available or if one is available, but in a terminal state, we try to retrieve a regular messenger.
3. If we can't retrieve a messenger or if the retrieved messenger is in a terminal state, an `IOException` is thrown.
4. Otherwise, the `sendMessageB()` method of the Messenger is called.

Bidirectional Pipe Communication

Bidirectional pipe connection requires the creation of a `JxtaServerPipe` on a server peer and to retrieve a `JxtaBidiPipe` using the accept method. The sending peer creates a `JxtaBidiPipe` using the same pipe advertisement as the server pipe to send messages.

New JxtaServerPipe

When the `JxtaServerPipe` is created, it creates an input pipe from the pipe service and behaves as a listener for it. When a new pipe connect message arrives, it used to create a `JxtaServerPipe` which is registered in a connection queue. During this process, a `Messenger` is retrieved from the endpoint service and used as a parameter to a protected constructor of the `JxtaBidiPipe`.

When the `accept()` method is called, the `JxtaServerPipe` polls the connection queue to retrieve an instance of a `JxtaBidiPipe` connection.

New JxtaBibiPipe

When a `JxtaBidiPipe` is created with one of its public constructor, it calls it `connect()` method which creates an open message (i.e., connection request) and an output pipe. Then, it waits for its resolution and notify listeners, if any.

Bidirectional pipes can be made reliable and use a “sliding window” mechanism (similar to TCP/IP) where received messages are acknowledged and re-ordered for proper delivery. These can be resent if necessary.

JxtaSocket & JxtaSocketServer

In order to establish a JxtaSocket communication, a 'server' peer will create a `JxtaSocketServer` and 'accept' a socket from it. Another 'socket' peer will create a `JxtaSocket`, using the same pipe advertisement as the 'server' peer.

When a JxtaSocket is created:

1. The default timeout for connection and closing is set a 60 seconds. The retry timeout is also set at 60 seconds.
2. A local input pipe is created and a connection is attempted. If the socket is to be reliable, a pair of `ReliableInputStream` and `ReliableOutputStream` are created, else a pair of `JxtaSocketInputStream` and `JxtaSocketOutputStreams` are created.

JxtaMulticastSocket

The `JxtaMulticastSocket` class replicates the typical UDP multicasting functionality over “groups” (i.e., reserved IP addresses), over peergroups using output pipes (unicast for communication to a specific peer, or propagate type).

However, `JxtaMulticastSocket` does not rely on UDP to communicate with other peers.

PeerInfo Service

We will not describe this service here as documents are available from: https://jxse-metering.dev.java.net/.

Membership and Access Services

These are the PSE Membership Service, the Non Membership Service, the Always Access Service, the Password Membership Service, the PSE AccessService and the Simple ACL Access Service. Unfortunately, we will not describe these in this document, since major revision and improvement of the code is expected in future releases of JXSE.

About The Future

This section contains comments and considerations about the future of JXSE.

The JXSE Architecture

OSGi, Modules, Service & JxtaLoader

Currently, the `RefJxtaLoader` extends the `JxtaLoader`. And, the `JxtaLoader` extends the `URLClassLoader` in the API code, which restricts other implementations of the JXTA service loader concept. Using `.jar` files as the base method to load services on a Java/JXTA platform has several drawbacks[13]:

- There is no way to indicate dependencies between `.jar` files.
- There is no versioning in `.jar` files. Hence, it is not possible load different versions simultaneously. Moreover, it is not possible to indicate which classes should be loaded when several versions are available and required by classes in other `.jars`.
- `.jar` files can contain conflicting names. Thus loading classes from classpath entries can cause unexpected and hard to solve conflicts.

This makes modular loading (of services) very hard to achieve on a large scale. Currently, JXSE dodges many of these issues, because all required modules and services are located in the delivered `jxta.jar` and few members of the community have implemented their own modules.

Suggestions

The OSGi framework solves all the `.jar` drawbacks. Therefore, it is in the interest of the community to pursue effort towards integration with OSGi. Early investigation indicate that re-implementing core JXTA service on top of the OSGi framework would probably be overkill, but nudging developers to implement JXTA services as OSGi services/bundles might be wise.

For example, an OSGi peer factory service could be implemented, from which peer objects could be used to create/join peer groups, on which OSGi/JXTA services could be loaded. These could be implemented via a delivered API interface or abstract class. Apache Felix iPOJO[14] may help facilitating coding and development.

Another benefit of OSGi is that offers simple functionality to check which other services have been loaded, and makes them easily accessible.

The `RefJxtaLoader` should progressively be deprecated and replaced by this alternative load mechanism. Corresponding peer group methods should be implemented.

Multiple peers in same JVM & Test Driven Implementation

One of the weaknesses of the JXSE implementation of JXTA protocols is that it does not allow the creation of multiple peers in the same JVM. This makes automatic testing of peer behavior nearly impossible to accomplish. There is a possibility of working around this issue by using separate class loaders, but this is cumbersome and very complex to achieve. Another solution, implemented by some, is to use mock classes.

The culprit is, amongst others, the peer group global registry (see `PeerGroup.GlobalRegistry`) which forces the creation of one and only one instance of a peergroup object per `PeerGroupIDs`. Other singletons may obstruct the creation of several peers within the same JVM.

Multiple peers in the same JVM has been requested by the community for quite some time, but requires some deeps modifications in core code. Its implementation would significantly facilitate a test driven continuous implementation process of JXSE.

Simplifying The Transportation Layer

The implementation of the Endpoint service and underlying transportation layer could be significantly simplified. Channel messengers are not really necessary and could be replaced by canonical messengers only. Direct messengers have been deactivated and should be removed from the code.

13 This well described in Neil Bartlett's 'OSGi in Practice' book preview (section 1.2), see http://neilbartlett.name/blog/osgibook/.
14 See http://felix.apache.org/site/apache-felix-ipojo.html.

Earlier leaders of the community have considered getting rid of blocking messengers, then asynchronous messengers and at last of channel messengers in the past. JXSE implements too many classes to achieve what has to be achieved. This consumes unnecessary resources (memory and processing time).

A lot of code has been written around 'hint processing', but these are only used in the Endpoint router and by immediate messengers (which have been deactivated). Many methods take in unnecessary parameters and propagate content to other methods. The code could be simplified.

Right now, `EndpointService` object instances do not share there connections to other peers (TCP, http, multicasting, etc...) between themselves. This is a unnecessary resource consumption. Since these are purely 'physical' means to exchange messages between peers, one could centralize them in a separate object to which peergroups would refer to to send messages and/or register listeners. In other words, we could have unique instances of `MessageTransport` objects.

Authentication & Security

Quite some code has already been written in order to implement security and authentication in the JXSE project, for example the `Credentials` and `Cjbx` objects. However, this has never been finalized. This allows anyone to publish advertisements and/or false routes to remote peers. Other key message exchanges (SRDI, leases, discovery queries...) are not protected and can be tampered with by third parties.

A member of the community has implemented the missing secured code. A big patch is available, but we have been missing workforce and time to test it and include in release 2.6.

UDP & Connectivity

Although JXSE uses datagrams to exchange messages via multicasting, it does not implement UDP. This transportation could be useful to send fast data between peers and to 'punch holes in NATs' à la Skype. Indeed, the relay seed mechanisms facilitates the access to peers located behind NATs, but forces the traffic through relay peers.

With UDP:

1. Peers N behind NATs could access peers W accessible on the WAN with UDP.
2. Peers W would register the translated IP address from N peers in their cache managers (as route advertisements).
3. Other remote peers O could fetch route advertisements from W peers and use the translated address to directly connect to N peers using UDP, assuming that N peers send frequent keep-alive messages to WAN peers to keep the translation valid.

This translation mechanism cannot be used with TCP, because O would fail to connect to N using the translated address, but some solutions have been proposed to enable TCP connections between peers located behind NAT using facilitators[15]. This could also be an alternative solution to relay seeds.

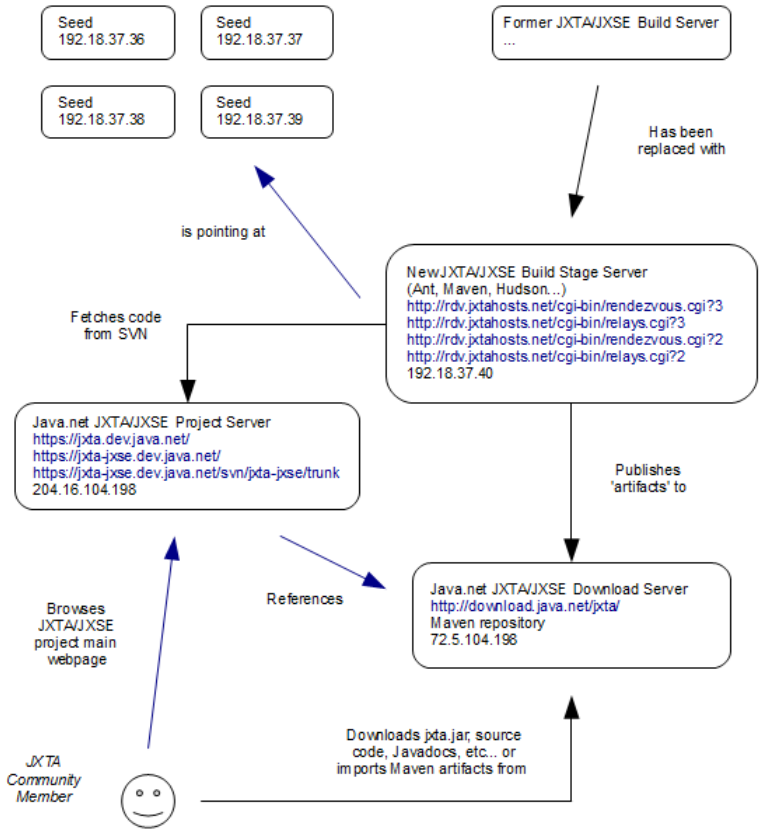
From Java 5 to Java 6

Release 2.6 is build with Java 5 to meet the Shoal project requirements, but several members in the community have requested to move to Java 6 as soon as possible. There is a high probability that release 2.6 will be the last using Java 5.

15 See http://reports-archive.adm.cs.cmu.edu/anon/isri2005/CMU-ISRI-05-104.pdf

Release Infrastructure & Support Environment

Current Infrastructure



At the end of 2008, the server used by the JXTA/JXSE community crashed. Some backup data could be restored, but the release system integrated with Java.net was broken. A new build stage server has been made available by Sun Microsystems for the community. Amongst others, this is where the list of default seeds and rendezvous relays could be fetched.

The idea was to retrieve (i.e., check-out) code from the Subversion repository on the Java.net JXTA/JXSE project server, build artifacts on the new build stage server and publish them on the Java.net JXTA/JXSE download server. Then users, browsing the main JXTA/JXSE web pages from the project server, would be redirected to the download server to download `jxta.jar`, code, documentation etc... for release 2.6, like they do for release 2.5 now.

Unfortunately, a) the new build stage server turns out to have some OpenSolaris installation issues which cannot be solved without the help of a system administrator and no support could be obtained to solve this issue, b) restoring the release process to `Java.net` is technically complex and subject to failures requiring access to the server to solve them and a good knowledge of OpenSolaris which does not seem to be available in the community, and c) the support, functionalities and web server response time offered by `Java.net` have become too insufficient to support the communities' effort.

Default Community Seeds

Although the JXSE code still contains references to the default communities seeds, these have not been maintained and are not operational anymore.

Moving from Java.net to Kenaï

It has been decided to move the JXTA/JXSE project from Java.net to project Kenaï. This process has started in November 2009 and will go through early 2010, in collaboration with Kenaï support. Two subprojects have been created:

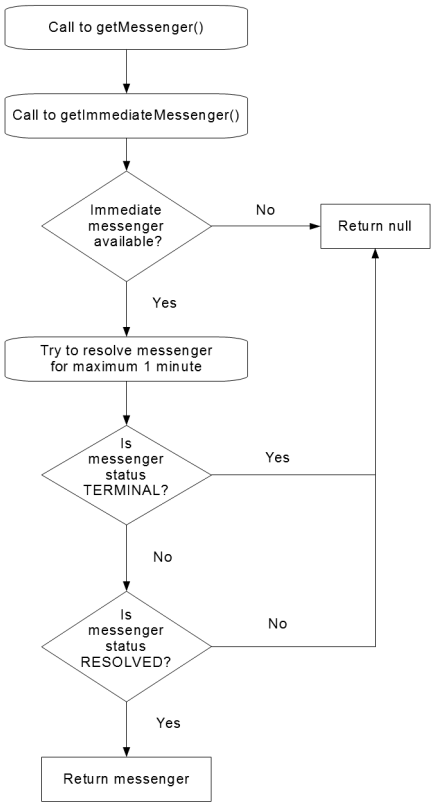
- JXTA - For the JXTA protocol specifications (http://kenai.com/projects/jxta).
- JXSE - For the Java implementation of the JXTA protocol specifications (http://kenai.com/projects/jxse). This is where the new subversion repository will be located and from which the code can be checked out.

Project Kenaï offers better functionality that Java.net, in an integrated environment, together with better support.

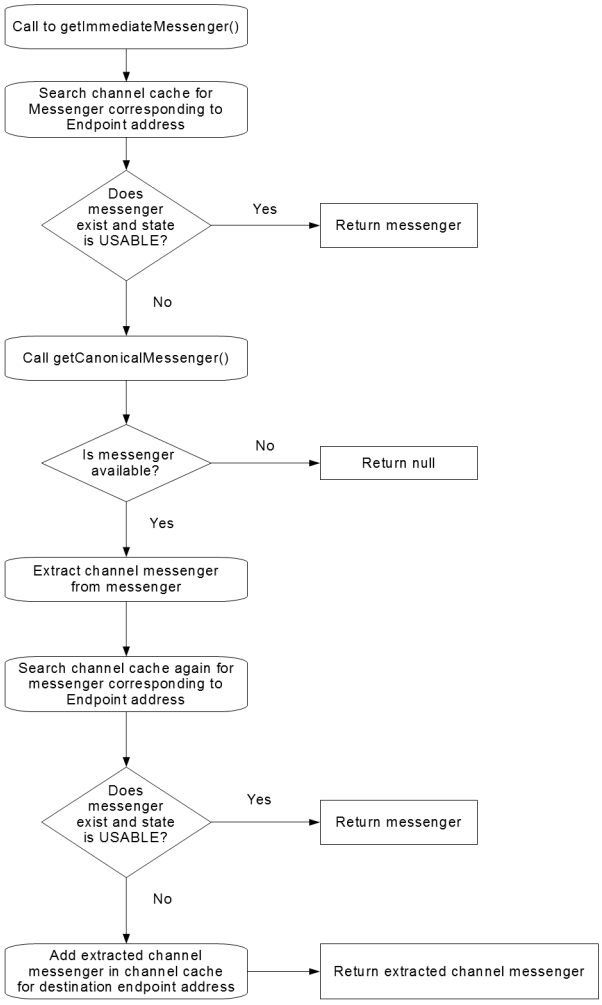
Appendix A: EndpointService Diagrams

This appendix contains diagrams helping understanding what is happening under the hood of JXSE.

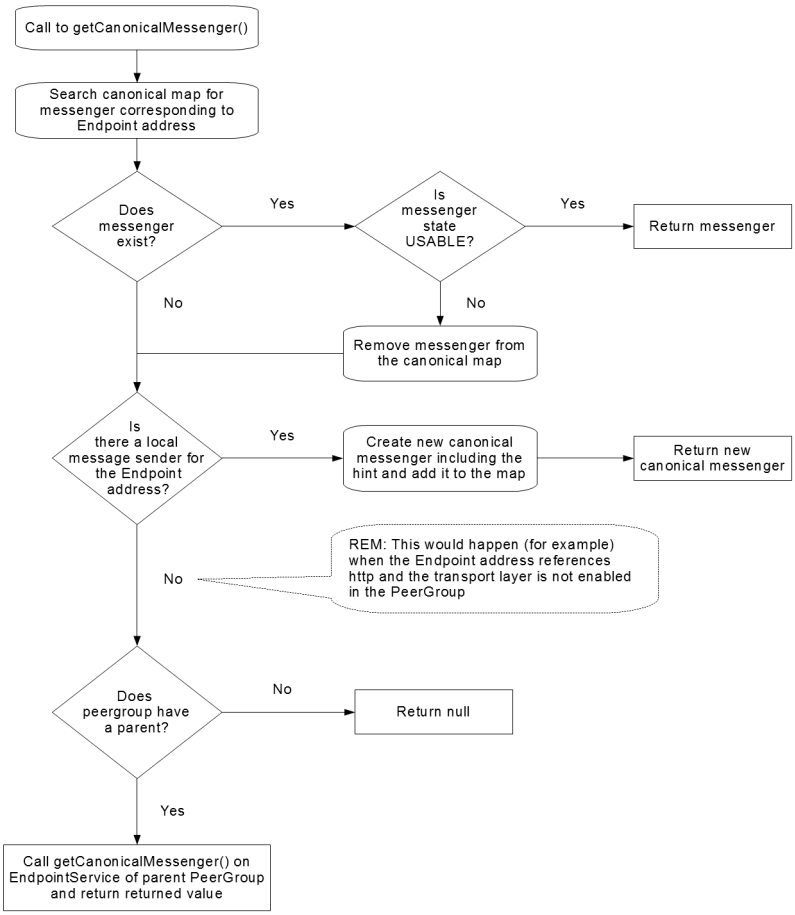
Endpoint Service – getMessenger()



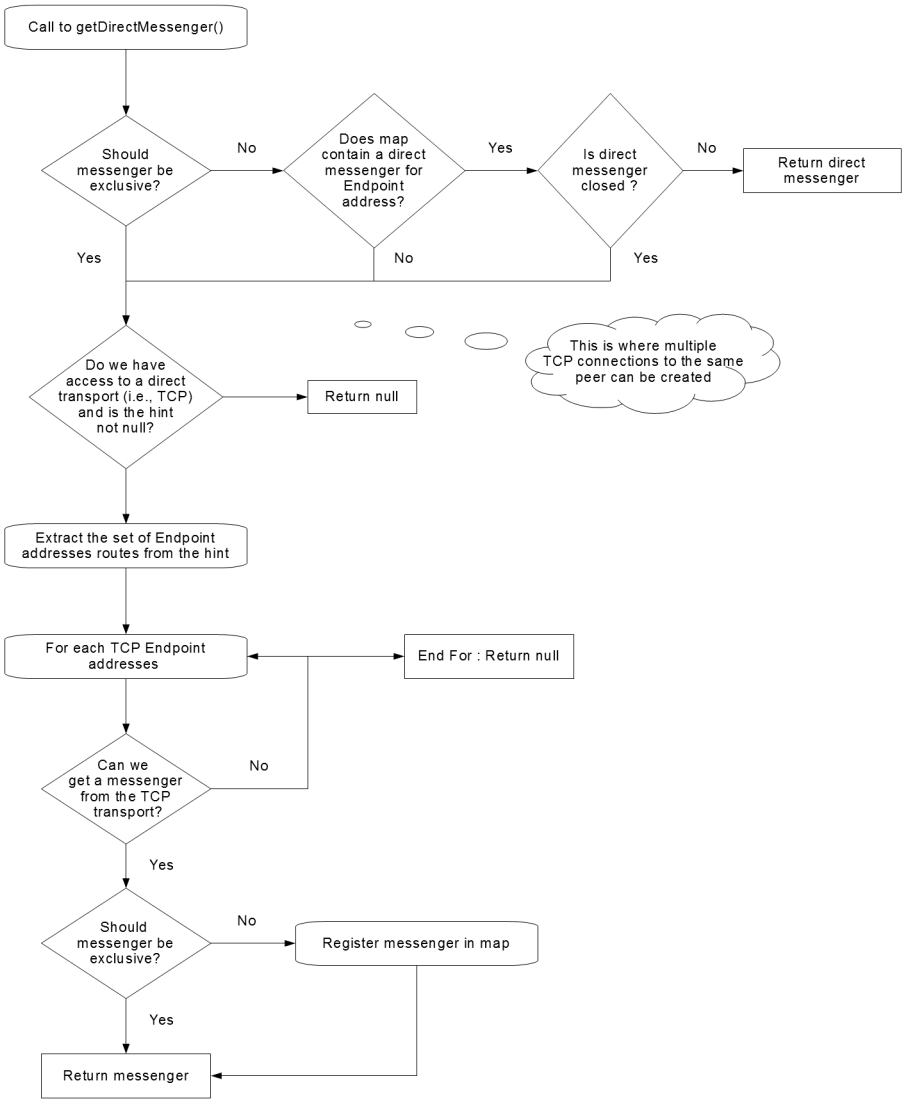
Endpoint Service – getMessengerImmediate()



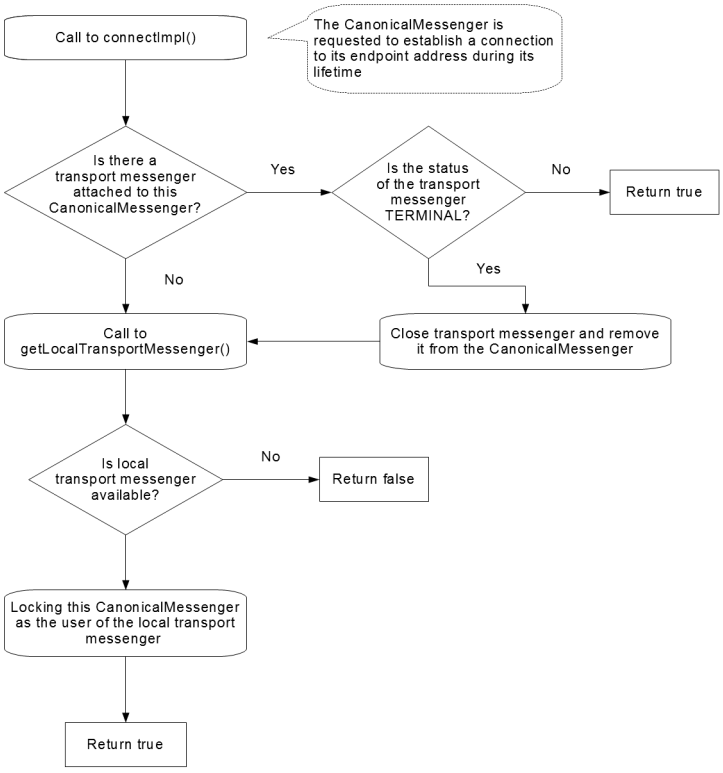
Endpoint Service – getCanonicalMessage()



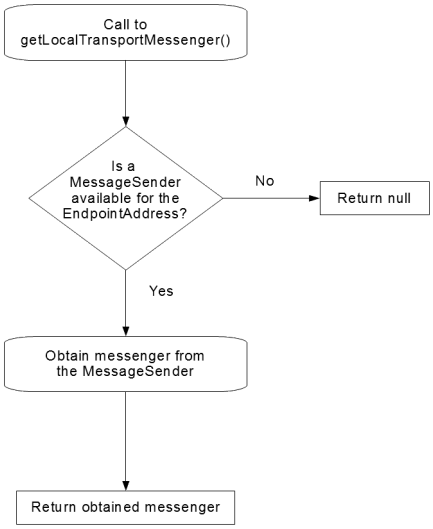
Endpoint Service – getDirectMessenger()



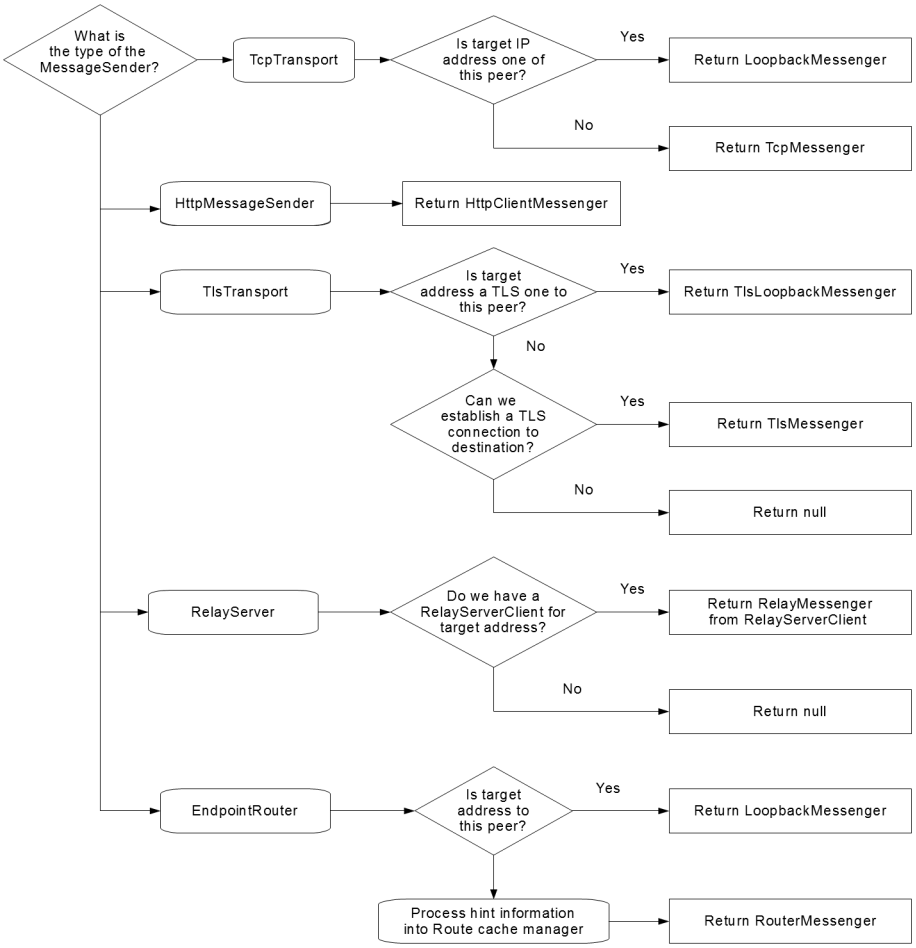
Canonical Messenger



Endpoint Service – getLocalTransportMessenger()

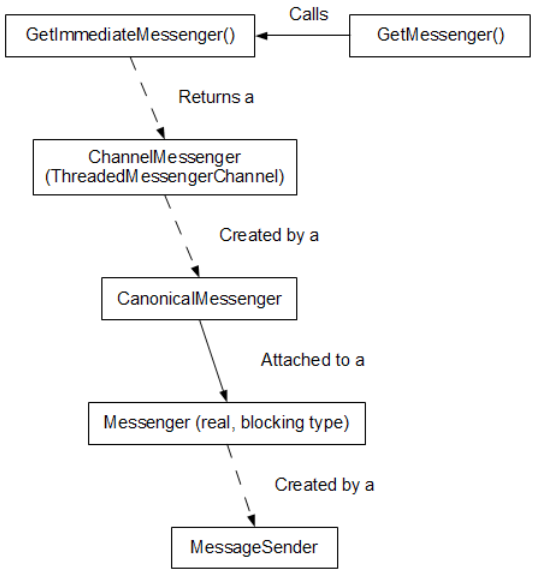


Message Sender – getMessenger()



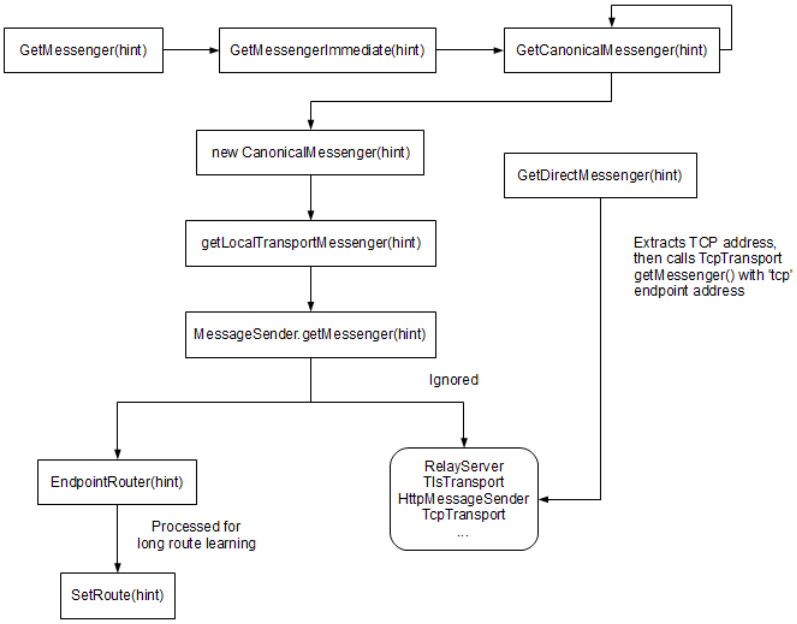
From Messengers to MessageSenders

This diagram shows the relationship between Messengers invoked via the EndpointService and MessageSenders physically sending messages:



Hint Processing

This diagram explains how hints (i.e., route suggestions) are taken into account when messengers are retrieved from the Endpoint service:

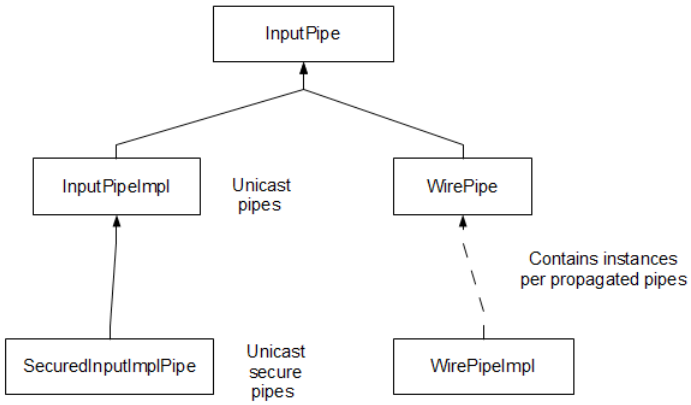


Appendix B: Pipe Diagrams

This appendix contains diagrams describing objects implementing pipes in JXSE.

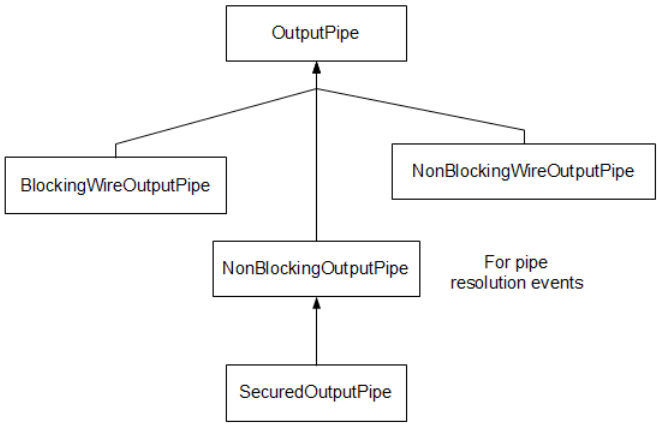
Input Pipe

Inheritance structure:



Output Pipe

Inheritance structure:



Appendix C: Connection Monitor Example

This appendix contains the complete code for the connection monitor example.

Connectivity Monitor

```
/**
 * This frame collects and displays connectivity information from a peergroup.
 */
public class ConnectivityMonitor extends JFrame implements Runnable {

    // Static

    public static final ScheduledExecutorService TheExecutor = Executors.newScheduledThreadPool(5);

    // Attributes

    private final JFrame ThisFrame;

    private PeerGroup ThePeerGroup = null;

    private Future TheMonitorFuture = null;

    private DefaultTableModel LocalRDVs_TM = null;
    private String[] LocalRDV_Col = { "Local RDV View IDs" };

    private DefaultTableModel LocalEdges_TM = null;
    private String[] LocalEdge_Col = { "Local EDGE View IDs" };

    private static final String[][] EmptyTableContent = new String[0][1];

    private ArrayList<LogEntry> TheLogs = new ArrayList<LogEntry>();

    private static final SimpleDateFormat TheDateFormat =
        new SimpleDateFormat("'['HH:mm:ss']'");

    /** Creates new form ConnectivityMonitor */
    public ConnectivityMonitor(PeerGroup inGroup) {

        // Registering as rendezvous event listener
        inGroup.getRendezVousService().addListener(new RdvEventMonitor(this));

        // Registering this JFrame
        ThisFrame = this;

        // JFrame initialization
        initComponents();

        // Displaying the frame on the awt queue
        SetDefaultLookAndFeel();
        putScreenAtTheCenter(this);
        resettingFrameValues();

        java.awt.EventQueue.invokeLater(new Runnable() {
            public void run() {
                ThisFrame.setVisible(true);
            }
        });

        // Initialization
        ThePeerGroup = inGroup;
```

```
        // Setting own default table models
        LocalRDVs_TM = new DefaultTableModel(EmptyTableContent, LocalRDV_Col);
        this.LocalRDVTable.setModel(LocalRDVs_TM);

        LocalEdges_TM = new DefaultTableModel(EmptyTableContent, LocalEdge_Col);
        this.LocalEdgeTable.setModel(LocalEdges_TM);

        // Starting the monitor
        TheLogs.add(new LogEntry(new Date(System.currentTimeMillis()),
            "Starting to monitor the peergroup"));

        TheMonitorFuture = TheExecutor.scheduleWithFixedDelay(this, 0, 1, TimeUnit.SECONDS);

    }

    public static void SetDefaultLookAndFeel() {

        try {
            UIManager.setLookAndFeel(UIManager.getSystemLookAndFeelClassName());
        } catch (ClassNotFoundException Ex) {
            System.err.println(Ex.toString());
        } catch (InstantiationException Ex) {
            System.err.println(Ex.toString());
        } catch (IllegalAccessException Ex) {
            System.err.println(Ex.toString());
        } catch (UnsupportedLookAndFeelException Ex) {
            System.err.println(Ex.toString());

        }

    }

    public static void putScreenAtTheCenter(Component TheComponent) {

        // Retrieving horizontal value
        int WidthPosition = (Toolkit.getDefaultToolkit().getScreenSize().width
                - TheComponent.getWidth()) / 2;

        int HeightPosition = (Toolkit.getDefaultToolkit().getScreenSize().height
                - TheComponent.getHeight()) / 2;

        TheComponent.setLocation(WidthPosition, HeightPosition);

    }

    /** This method is called from within the constructor to
     * initialize the form.
     * WARNING: Do NOT modify this code. The content of this method is
     * always regenerated by the Form Editor.
     */
    @SuppressWarnings("unchecked")
    // <editor-fold defaultstate="collapsed" desc="Generated Code">
    private void initComponents() {

        StatusPane = new javax.swing.JPanel();
        PeerIDLabel = new javax.swing.JLabel();
        PeerNameTextField = new javax.swing.JTextField();
        PeerGroupIDLabel = new javax.swing.JLabel();
        PeerGroupNameTextField = new javax.swing.JTextField();
        ParentGroupLabel = new javax.swing.JLabel();
        ParentGroupNameTextField = new javax.swing.JTextField();
        PeerIDTextField = new javax.swing.JTextField();
```

```
        PeerGroupIDTextField = new javax.swing.JTextField();
        ParentGroupIDTextField = new javax.swing.JTextField();
        ScrollLogPane = new javax.swing.JScrollPane();
        LogPane = new javax.swing.JTextPane();
        DisplayPanel = new javax.swing.JPanel();
        jScrollPane1 = new javax.swing.JScrollPane();
        LocalEdgeTable = new javax.swing.JTable();
        jScrollPane2 = new javax.swing.JScrollPane();
        LocalRDVTable = new javax.swing.JTable();
        IsConnectedToRelayTextField = new javax.swing.JTextField();
        IsConnectedToRelayCheckBox = new javax.swing.JCheckBox();
        IsConnectedToRDVCheckBox = new javax.swing.JCheckBox();
        CurrentModeLabel2 = new javax.swing.JLabel();
        AliveRadioButton = new javax.swing.JRadioButton();
        IsRDVCheckBox = new javax.swing.JCheckBox();

        setDefaultCloseOperation(javax.swing.WindowConstants.EXIT_ON_CLOSE);
        setResizable(false);
        addWindowListener(new java.awt.event.WindowAdapter() {
            public void windowClosed(java.awt.event.WindowEvent evt) {
                formWindowClosed(evt);
            }
        });

        PeerIDLabel.setFont(new java.awt.Font("Tahoma", 1, 11));
        PeerIDLabel.setText("Peer");

        PeerNameTextField.setEditable(false);
        PeerNameTextField.setFont(new java.awt.Font("Tahoma", 0, 9));

        PeerGroupIDLabel.setFont(new java.awt.Font("Tahoma", 1, 11));
        PeerGroupIDLabel.setText("Peer Group");

        PeerGroupNameTextField.setEditable(false);
        PeerGroupNameTextField.setFont(new java.awt.Font("Tahoma", 0, 9));

        ParentGroupLabel.setFont(new java.awt.Font("Tahoma", 1, 11));
        ParentGroupLabel.setText("Parent Group");

        ParentGroupNameTextField.setEditable(false);
        ParentGroupNameTextField.setFont(new java.awt.Font("Tahoma", 0, 9));

        PeerIDTextField.setEditable(false);
        PeerIDTextField.setFont(new java.awt.Font("Tahoma", 0, 9));

        PeerGroupIDTextField.setEditable(false);
        PeerGroupIDTextField.setFont(new java.awt.Font("Tahoma", 0, 9));

        ParentGroupIDTextField.setEditable(false);
        ParentGroupIDTextField.setFont(new java.awt.Font("Tahoma", 0, 9));

        LogPane.setEditable(false);
        LogPane.setFont(new java.awt.Font("Tahoma", 0, 9));
        LogPane.setName("LogPane"); // NOI18N
        ScrollLogPane.setViewportView(LogPane);

        javax.swing.GroupLayout StatusPaneLayout = new javax.swing.GroupLayout(StatusPane);
        StatusPane.setLayout(StatusPaneLayout);
        StatusPaneLayout.setHorizontalGroup(
            StatusPaneLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                                            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
StatusPaneLayout.createSequentialGroup()
```

```
                .addContainerGap()
                .addGroup(StatusPaneLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.TRAI
LING)
                            .addComponent(ScrollLogPane, javax.swing.GroupLayout.Alignment.LEADING,
javax.swing.GroupLayout.DEFAULT_SIZE, 401, Short.MAX_VALUE)
                        .addComponent(PeerGroupIDTextField, javax.swing.GroupLayout.Alignment.LEADING,
javax.swing.GroupLayout.DEFAULT_SIZE, 401, Short.MAX_VALUE)
                            .addComponent(PeerIDTextField, javax.swing.GroupLayout.DEFAULT_SIZE, 401,
Short.MAX_VALUE)
                                                    .addGroup(javax.swing.GroupLayout.Alignment.LEADING,
StatusPaneLayout.createSequentialGroup()
                        .addComponent(PeerIDLabel)
                        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
                          .addComponent(PeerNameTextField, javax.swing.GroupLayout.DEFAULT_SIZE, 365,
Short.MAX_VALUE))
                                                    .addGroup(javax.swing.GroupLayout.Alignment.LEADING,
StatusPaneLayout.createSequentialGroup()
                        .addComponent(ParentGroupLabel)
                        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
                        .addComponent(ParentGroupNameTextField, javax.swing.GroupLayout.DEFAULT_SIZE,
316, Short.MAX_VALUE))
                                                    .addGroup(javax.swing.GroupLayout.Alignment.LEADING,
StatusPaneLayout.createSequentialGroup()
                        .addComponent(PeerGroupIDLabel)
                        .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
                          .addComponent(PeerGroupNameTextField, javax.swing.GroupLayout.DEFAULT_SIZE,
328, Short.MAX_VALUE))
                    .addComponent(ParentGroupIDTextField, javax.swing.GroupLayout.Alignment.LEADING,
javax.swing.GroupLayout.DEFAULT_SIZE, 401, Short.MAX_VALUE))
                .addContainerGap())
        );
        StatusPaneLayout.setVerticalGroup(
            StatusPaneLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(StatusPaneLayout.createSequentialGroup()
                .addContainerGap()
                .addGroup(StatusPaneLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASE
LINE)
                    .addComponent(PeerIDLabel)
                            .addComponent(PeerNameTextField, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                             .addComponent(PeerIDTextField, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                .addGroup(StatusPaneLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASE
LINE)
                    .addComponent(PeerGroupIDLabel)
                        .addComponent(PeerGroupNameTextField, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                          .addComponent(PeerGroupIDTextField, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                .addGroup(StatusPaneLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BASE
LINE)
                    .addComponent(ParentGroupLabel)
                      .addComponent(ParentGroupNameTextField, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                        .addComponent(ParentGroupIDTextField, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(18, 18, 18)
                            .addComponent(ScrollLogPane, javax.swing.GroupLayout.DEFAULT_SIZE, 269,
Short.MAX_VALUE))
        );

        LocalEdgeTable.setFont(new java.awt.Font("Tahoma", 0, 9));
        LocalEdgeTable.setModel(new javax.swing.table.DefaultTableModel(
```

```
            new Object [][] {
                {null}
            },
            new String [] {
                "Local Edge View IDs"
            }
        ) {
            Class[] types = new Class [] {
                java.lang.String.class
            };
            boolean[] canEdit = new boolean [] {
                false
            };

            public Class getColumnClass(int columnIndex) {
                return types [columnIndex];
            }

            public boolean isCellEditable(int rowIndex, int columnIndex) {
                return canEdit [columnIndex];
            }
        });
        jScrollPane1.setViewportView(LocalEdgeTable);
        LocalEdgeTable.getColumnModel().getColumn(0).setResizable(false);

        LocalRDVTable.setFont(new java.awt.Font("Tahoma", 0, 9));
        LocalRDVTable.setModel(new javax.swing.table.DefaultTableModel(
            new Object [][] {
                {null}
            },
            new String [] {
                "Local RDV View IDs"
            }
        ) {
            Class[] types = new Class [] {
                java.lang.String.class
            };
            boolean[] canEdit = new boolean [] {
                false
            };

            public Class getColumnClass(int columnIndex) {
                return types [columnIndex];
            }

            public boolean isCellEditable(int rowIndex, int columnIndex) {
                return canEdit [columnIndex];
            }
        });
        jScrollPane2.setViewportView(LocalRDVTable);
        LocalRDVTable.getColumnModel().getColumn(0).setResizable(false);

        IsConnectedToRelayTextField.setEditable(false);
        IsConnectedToRelayTextField.setFont(new java.awt.Font("Tahoma", 0, 9));

        IsConnectedToRelayCheckBox.setFont(new java.awt.Font("Tahoma", 1, 11));
        IsConnectedToRelayCheckBox.setText("is connected to Relay");
        IsConnectedToRelayCheckBox.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                IsConnectedToRelayCheckBoxActionPerformed(evt);
            }
        });
```

```
        IsConnectedToRDVCheckBox.setFont(new java.awt.Font("Tahoma", 1, 11));
        IsConnectedToRDVCheckBox.setText("is connected to RDV");
        IsConnectedToRDVCheckBox.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                IsConnectedToRDVCheckBoxActionPerformed(evt);
            }
        });

        CurrentModeLabel2.setFont(new java.awt.Font("Tahoma", 1, 11));
        CurrentModeLabel2.setText("Relay ID");

        AliveRadioButton.setFont(new java.awt.Font("Tahoma", 1, 11));
        AliveRadioButton.setText("Alive");

        IsRDVCheckBox.setFont(new java.awt.Font("Tahoma", 1, 11));
        IsRDVCheckBox.setText("is RDV");
        IsRDVCheckBox.addActionListener(new java.awt.event.ActionListener() {
            public void actionPerformed(java.awt.event.ActionEvent evt) {
                IsRDVCheckBoxActionPerformed(evt);
            }
        });

        javax.swing.GroupLayout DisplayPanelLayout = new javax.swing.GroupLayout(DisplayPanel);
        DisplayPanel.setLayout(DisplayPanelLayout);
        DisplayPanelLayout.setHorizontalGroup(
            DisplayPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(DisplayPanelLayout.createSequentialGroup()
                .addContainerGap()
                .addGroup(DisplayPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LE
ADING)
                          .addComponent(jScrollPane2, javax.swing.GroupLayout.DEFAULT_SIZE, 372,
Short.MAX_VALUE)
                          .addComponent(jScrollPane1, javax.swing.GroupLayout.DEFAULT_SIZE, 372,
Short.MAX_VALUE)
                    .addComponent(IsConnectedToRelayTextField, javax.swing.GroupLayout.DEFAULT_SIZE,
372, Short.MAX_VALUE)
                                                .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
DisplayPanelLayout.createSequentialGroup()
                        .addGroup(DisplayPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alig
nment.LEADING)
                            .addComponent(CurrentModeLabel2)
                            .addComponent(AliveRadioButton))
                             .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED, 93,
Short.MAX_VALUE)
                        .addComponent(IsRDVCheckBox)
                        .addGap(18, 18, 18)
                        .addGroup(DisplayPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alig
nment.LEADING)
                            .addComponent(IsConnectedToRDVCheckBox)
                            .addComponent(IsConnectedToRelayCheckBox))))
                .addContainerGap())
        );
        DisplayPanelLayout.setVerticalGroup(
            DisplayPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
                                                .addGroup(javax.swing.GroupLayout.Alignment.TRAILING,
DisplayPanelLayout.createSequentialGroup()
                .addContainerGap()
                .addGroup(DisplayPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BA
SELINE)
                    .addComponent(IsConnectedToRDVCheckBox)
                    .addComponent(AliveRadioButton)
                    .addComponent(IsRDVCheckBox))
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.UNRELATED)
```

```
                .addGroup(DisplayPanelLayout.createParallelGroup(javax.swing.GroupLayout.Alignment.BA
SELINE)
                    .addComponent(IsConnectedToRelayCheckBox, javax.swing.GroupLayout.PREFERRED_SIZE,
23, javax.swing.GroupLayout.PREFERRED_SIZE)
                        .addComponent(CurrentModeLabel2, javax.swing.GroupLayout.PREFERRED_SIZE, 17,
javax.swing.GroupLayout.PREFERRED_SIZE))
                            .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED,  6,
Short.MAX_VALUE)
                    .addComponent(IsConnectedToRelayTextField, javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(18, 18, 18)
                            .addComponent(jScrollPane2,  javax.swing.GroupLayout.PREFERRED_SIZE,  111,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addGap(18, 18, 18)
                            .addComponent(jScrollPane1,  javax.swing.GroupLayout.PREFERRED_SIZE,  192,
javax.swing.GroupLayout.PREFERRED_SIZE)
                .addContainerGap())
        );

        javax.swing.GroupLayout layout = new javax.swing.GroupLayout(getContentPane());
        getContentPane().setLayout(layout);
        layout.setHorizontalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(javax.swing.GroupLayout.Alignment.TRAILING, layout.createSequentialGroup()
                                    .addComponent(StatusPane,  javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                .addPreferredGap(javax.swing.LayoutStyle.ComponentPlacement.RELATED)
                                .addComponent(DisplayPanel,  javax.swing.GroupLayout.PREFERRED_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, javax.swing.GroupLayout.PREFERRED_SIZE))
        );
        layout.setVerticalGroup(
            layout.createParallelGroup(javax.swing.GroupLayout.Alignment.LEADING)
            .addGroup(layout.createSequentialGroup()
                .addGap(11, 11, 11)
                                  .addComponent(DisplayPanel,  javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE))
            .addGroup(layout.createSequentialGroup()
                                    .addComponent(StatusPane,  javax.swing.GroupLayout.DEFAULT_SIZE,
javax.swing.GroupLayout.DEFAULT_SIZE, Short.MAX_VALUE)
                .addGap(11, 11, 11))
        );

        pack();
    }// </editor-fold>

    private void IsConnectedToRelayCheckBoxActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
    }

    private void formWindowClosed(java.awt.event.WindowEvent evt) {

        // Stopping monitor task
        stopMonitorTask();

    }

    private void IsConnectedToRDVCheckBoxActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
    }

    private void IsRDVCheckBoxActionPerformed(java.awt.event.ActionEvent evt) {
        // TODO add your handling code here:
    }

    private synchronized void stopMonitorTask() {
```

```
        if ( TheMonitorFuture != null ) {
            TheMonitorFuture.cancel(false);
        }

    }

    @Override
    protected void finalize() {

        // Stopping monitor task
        stopMonitorTask();

    }

    // Variables declaration - do not modify
    private javax.swing.JRadioButton AliveRadioButton;
    private javax.swing.JLabel CurrentModeLabel2;
    private javax.swing.JPanel DisplayPanel;
    private javax.swing.JCheckBox IsConnectedToRDVCheckBox;
    private javax.swing.JCheckBox IsConnectedToRelayCheckBox;
    private javax.swing.JTextField IsConnectedToRelayTextField;
    private javax.swing.JCheckBox IsRDVCheckBox;
    private javax.swing.JTable LocalEdgeTable;
    private javax.swing.JTable LocalRDVTable;
    private javax.swing.JTextPane LogPane;
    private javax.swing.JTextField ParentGroupIDTextField;
    private javax.swing.JLabel ParentGroupLabel;
    private javax.swing.JTextField ParentGroupNameTextField;
    private javax.swing.JLabel PeerGroupIDLabel;
    private javax.swing.JTextField PeerGroupIDTextField;
    private javax.swing.JTextField PeerGroupNameTextField;
    private javax.swing.JLabel PeerIDLabel;
    private javax.swing.JTextField PeerIDTextField;
    private javax.swing.JTextField PeerNameTextField;
    private javax.swing.JScrollPane ScrollLogPane;
    private javax.swing.JPanel StatusPane;
    private javax.swing.JScrollPane jScrollPane1;
    private javax.swing.JScrollPane jScrollPane2;
    // End of variables declaration

    public void resettingFrameValues() {

        // Resetting frame value
        this.setTitle("Connectivity Monitor");

        this.PeerNameTextField.setText("<unknown>");
        this.PeerIDTextField.setText("");

        this.PeerGroupNameTextField.setText("<unknown>");
        this.PeerGroupIDTextField.setText("");

        this.ParentGroupNameTextField.setText("<unknown>");
        this.ParentGroupIDTextField.setText("");

        this.IsConnectedToRelayCheckBox.setSelected(false);
        this.IsConnectedToRDVCheckBox.setSelected(false);

    }

      private void updateTableContent(DefaultTableModel inTM, List<String> inNewContent, String[]
inColumns) {
```

```
        // Do we have the same number of elements
        if ( inTM.getRowCount() == inNewContent.size() ) {

            // Sorting new candidates
            Collections.sort(inNewContent);

            // Replacing items that have to be replaced
            for (int i=0;i<inNewContent.size();i++) {

                if ( inNewContent.get(i).compareTo((String) inTM.getValueAt(i, 0)) != 0 )
                    inTM.setValueAt(inNewContent.get(i), i, 0);

            }

            // Done
            return;

        }

        // We need a new data vector
        String[][] NewContent = new String[inNewContent.size()][1];
        for (int i=0;i<inNewContent.size();i++) NewContent[i][0] = inNewContent.get(i);
        inTM.setDataVector(NewContent, inColumns);

    }

    public void run() {

        // Alive notification
        this.AliveRadioButton.setSelected(!this.AliveRadioButton.isSelected());

        if ( this.ThePeerGroup == null ) {

            LogEntry TheEntry = new LogEntry(new Date(System.currentTimeMillis()),
                "Monitored peergroup is NULL");
            this.TheLogs.add(TheEntry);

            // Resetting frame value
            resettingFrameValues();

            return;

        }

        this.setTitle("Connectivity Monitor - "
                + ThePeerGroup.getPeerName() + " - "
                + ThePeerGroup.getPeerID().toString());

        this.PeerNameTextField.setText(ThePeerGroup.getPeerName());
        this.PeerIDTextField.setText(ThePeerGroup.getPeerID().toString());

        this.PeerGroupNameTextField.setText(ThePeerGroup.getPeerGroupName());
        this.PeerGroupIDTextField.setText(ThePeerGroup.getPeerGroupID().toString());

        PeerGroup ParentPG = this.ThePeerGroup.getParentGroup();

        if ( ParentPG != null) {

            this.ParentGroupNameTextField.setText(ParentPG.getPeerGroupName());
            this.ParentGroupIDTextField.setText(ParentPG.getPeerGroupID().toString());
```

```
        }

        RendezVousService TmpRDVS = this.ThePeerGroup.getRendezVousService();

        if ( TmpRDVS != null ) {

            this.IsRDVCheckBox.setSelected(TmpRDVS.isRendezVous());
            this.IsConnectedToRDVCheckBox.setSelected(TmpRDVS.isConnectedToRendezVous());

            List<PeerID> Items = TmpRDVS.getLocalRendezVousView();

            // Sorting Peer IDs
            List<String> StrItems = new ArrayList<String>();
            for (int i=0;i<Items.size();i++)
                StrItems.add(Items.get(i).toString());

            updateTableContent(LocalRDVs_TM, StrItems, LocalRDV_Col);

        } else {

            TheLogs.add(new LogEntry(new Date(System.currentTimeMillis()),
                "Rendezvous service is NULL"));

        }

        EndpointService TmpRelay = this.ThePeerGroup.getEndpointService();

        if ( TmpRelay != null ) {

            this.IsConnectedToRelayCheckBox.setSelected(TmpRelay.isConnectedToRelayPeer());

            List<PeerID> Items = TmpRDVS.getLocalEdgeView();

            // Sorting Peer IDs
            List<String> StrItems = new ArrayList<String>();
            for (int i=0;i<Items.size();i++)
                StrItems.add(Items.get(i).toString());

            updateTableContent(LocalEdges_TM, StrItems, LocalEdge_Col);

        } else {

            TheLogs.add(new LogEntry(new Date(System.currentTimeMillis()),
                "Endpoint service is NULL"));

        }

        // Adding logs

        String Content = "";
        Collections.sort(TheLogs);

        for (int i=(TheLogs.size()-1);i>=0;i--)
            Content = Content + TheDateFormat.format(TheLogs.get(i).TheDate)
                + " " + TheLogs.get(i).TheLog + "\n";

        LogPane.setText(Content);

    }

    private static class LogEntry implements Comparable<LogEntry> {
```

```
        public Date TheDate = null;
        public String TheLog = null;

        public LogEntry(Date inDate, String inLog) {

            TheDate = inDate;
            TheLog = inLog;

        }

        public int compareTo(LogEntry o) {

            if ( o == null ) return 1;

            return TheDate.compareTo(o.TheDate);

        }

    }

    public static class RdvEventMonitor implements RendezvousListener {

        private ConnectivityMonitor TheMonitor = null;

        public RdvEventMonitor(ConnectivityMonitor inCM) {

            TheMonitor = inCM;

        }

        public void rendezvousEvent(RendezvousEvent event) {

            if ( event == null ) return;

            Date TimeStamp = new Date(System.currentTimeMillis());
            String Log = null;

            if ( event.getType() == RendezvousEvent.RDVCONNECT ) {
                Log = "Connection to RDV";
            } else if ( event.getType() == RendezvousEvent.RDVRECONNECT ) {
                Log = "Reconnection to RDV";
            } else if ( event.getType() == RendezvousEvent.CLIENTCONNECT ) {
                Log = "EDGE client connection";
            } else if ( event.getType() == RendezvousEvent.CLIENTRECONNECT ) {
                Log = "EDGE client reconnection";
            } else if ( event.getType() == RendezvousEvent.RDVDISCONNECT ) {
                Log = "Disconnection from RDV";
            } else if ( event.getType() == RendezvousEvent.RDVFAILED ) {
                Log = "Connection to RDV failed";
            } else if ( event.getType() == RendezvousEvent.CLIENTDISCONNECT ) {
                Log = "EDGE client disconnection from RDV";
            } else if ( event.getType() == RendezvousEvent.CLIENTFAILED ) {
                Log = "EDGE client connection to RDV failed";
            } else if ( event.getType() == RendezvousEvent.BECAMERDV ) {
                Log = "This peer became RDV";
            } else if ( event.getType() == RendezvousEvent.BECAMEEDGE ) {
                Log = "This peer became EDGE";
            }

            String TempPID = event.getPeer();
            if ( TempPID != null ) Log = Log + "\n  " + TempPID;
```

```
            // Adding the entry
            TheMonitor.TheLogs.add(new LogEntry(TimeStamp, Log));

        }

    }

}
```

DelayedJxtaNetworkStopper

```
public class DelayedJxtaNetworkStopper implements Runnable {

    // Attributes

    private final NetworkManager TheNM;
    private String TheMsg = "";
    private String TheTitle = "";

    public DelayedJxtaNetworkStopper(NetworkManager inNM, String inMsg, String inTitle) {

        TheNM = inNM;
        TheMsg = inMsg;
        TheTitle = inTitle;

    }

    public void run() {

        // Creating the dialog box
        JOptionPane pane = new JOptionPane(TheMsg, JOptionPane.INFORMATION_MESSAGE);
        JDialog dialog = pane.createDialog(TheTitle);

        // Displaying the message in a non-modal way
        dialog.setModal(false);
        dialog.setVisible(true);

        // Waiting for user to click ok
        Object click = null;

        do {
            click = pane.getValue();
            if ( JOptionPane.UNINITIALIZED_VALUE == click )
                try { Thread.sleep(200); } catch (InterruptedException ex) {}
        } while ( click == JOptionPane.UNINITIALIZED_VALUE );

        // Closing dialog
        dialog.dispose();

        // Stopping the network
        TheNM.stopNetwork();

    }

}
```

Appendix D: OSGi Integration Study

Date: July 29th , 2009

Author: Jérôme Verstrynge

Introduction

This document analyzes the possibility of integrating OSGi with JXTA and JXSE. It is based on the Felix Apache Tutorial and a comprehensive OSGi introduction reading available at:

http://gravity.sourceforge.net/servicebinder/osginutshell.html

The purpose is to analyze whether OSGi can be integrated with JXTA/JXSE from architectural perspective and if so, what would be the obstacles to overcome and what would be the main steps towards reaching this objective.

Reminder

Modules in JXTA/JXSE

Terminology

JXTA defines some module terminology:

- *Module Class* – This is an abstraction to define a set of modules offering a similar set of functionality in JXTA. This abstraction is independent of any JXTA implementation (i.e. binding) or module implementations. 'Class' is not to be confused with the Java programming language `Class` concept.
- *Module Specification* – Considering a module class ID, a module specification ID designates a family of network-compatible implementation of the corresponding module class.

  REM: Typically, a C++ binding of JXTA could not share its module code with a Java binding, even though the modules could serve the same 'class' of purposes.

  However, each module can communicate with each other using a JXTA pipe regardless of their implementation code. They are network compatible.
- *Module Implementation* – Considering a module specification ID, a module implementation ID refers to one of the implementations of a module specification (in a specific programming language).

  REM: For example, a module class for cache data management - for a given binding - can be implemented different ways: with text files, with a database.

For each type of module, a corresponding advertisement is defined (module class advertisement, module specification advertisement and module implementation advertisement).

The module specification advertisement contains a mandatory version field and an optional JXTA pipe advertisement. The module implementation advertisement provides an optional URI where the code can be loaded.

JxtaLoader

The `JxtaLoader` is an JXSE API abstract class extending `URLClassLoader`, which extends `SecureClassLoader`, which extends `ClassLoader`. The `JxtaLoader` operates in a similar way to `ClassLoader`.

Amongst other, ClassLoaders define a couple of services:

- *Defining a Class* – From a class Java binary name and data, an instance of the `Class` object is created. It can later be used to create an instance of that class.
- *Finding a Class* – Returns an instance of the `Class` object from a class Java binary name. If this class cannot be found within the loader, the parent loader may be used.
- *Load a Class* – Returns an loaded instance of the `Class` object.

The `JxtaLoader` defines similar services to load JXTA modules:

- *Defining a Module Class* – From a a Module Implementation advertisement, an instance of a `Class` object implementing the JXTA/JXSE `Module` interface is created. It can later be used to create an instance of that `Module`.
- *Finding a Module Class* – Returns an instance of a `Class` object implementing the JXTA/JXSE `Module` interface from its Module Specification ID. If necessary, the class is loaded first.
- *Loading a Module Class* – Returns an loaded instance of a `Class` object implementing the JXTA/JXSE `Module` interface from its Module Specification ID. If necessary, the class is loaded using the provided URLs pointing to JARs.
- *Finding Module Implementation Advertisement* – Returns a Module Implementation Advertisement from a `Class` object implementing the JXTA/JXSE `Module` interface or from a Module Specification ID.

JXSE JxtaLoader

The JXSE implementation of the JXTA loader is based on the JAR Service Provider concept. Currently, the delivered `jxta.jar` file contains a mapping between Module Specifications Ids and Java Implementation classes.

JXSE Module Life-cycle

All JXSE modules extend the `Module` interface, which defines 3 methods: `init`, `startApp` and `StopApp`. The first will initialize the module. The second will start it, but may stall while waiting for another module to start first. It may have to be called again. The last method is invoked to stop the module.

Identified Issues & Limitations

- JXTA specifications 'understand' that modules are related to peergroups. This is conceptually limiting, but has no real impact on possible implementations the module concept.
- JXTA does not define a relationship between module specification advertisements and corresponding module class Ids (hierarchy or 1-n relationship).
- JXTA does not define a relationship between module implementation advertisements and corresponding module specification Ids (hierarchy or 1-n relationship).
- JXSE does not implement module implementation Ids and makes direct links between module specification Ids and Java classes.
- JXSE defines default Module Class Ids in the code, but not their corresponding module class advertisement.
- JXSE defines default Module Specification Ids in the code, but not their corresponding module specification advertisements.

OSGi

Bundle

The fundamental unit in OSGi is a *bundle*, that is, a Jar file with a `MANIFEST.MF` file. This bundle contains *services* objects, that is, objects implementing some delivered interfaces defining methods called by OSGi.

For example:

```
import org.osgi.framework.BundleActivator;
import org.osgi.framework.BundleContext;
import org.osgi.framework.ServiceListener;
import org.osgi.framework.ServiceEvent;

public class Activator implements BundleActivator, ServiceListener {

    // Implements BundleActivator.start()

    public void start(BundleContext context) {
        System.out.println("Starting to listen for service events.");
        context.addServiceListener(this);
    }

    // Implements BundleActivator.stop()

    public void stop(BundleContext context) {
        context.removeServiceListener(this);
        System.out.println("Stopped listening for service events.");
    }

    // Implements ServiceListener.serviceChanged()

    public void serviceChanged(ServiceEvent event) {

        String[] objectClass = (String[])
            event.getServiceReference().getProperty("objectClass");

        if (event.getType() == ServiceEvent.REGISTERED) {
            System.out.printl("Service registered");
        } else if (event.getType() == ServiceEvent.UNREGISTERING) {
            System.out.printl("Service unregistered");
        } else if (event.getType() == ServiceEvent.MODIFIED) {
            System.out.printl("Service modified");
        }

    }

}
```

The above is an example of bundle code. The `start()` method is called when the bundle reaches the STARTING status in its life cycle. The `stop()` method is called when the bundle reaches the STOP status in its life cycle.

The following is an example of a `MANIFEST.MF` file for a bundle:

```
Bundle-Name: Hello World
Bundle-SymbolicName: org.wikipedia.helloworld
Bundle-Description: A Hello World bundle
Bundle-ManifestVersion: 2
Bundle-Version: 1.0.0
Bundle-Activator: org.wikipedia.Activator
Export-Package: org.wikipedia.helloworld;version="1.0.0"
Import-Package: org.osgi.framework;version="1.3.0"
```

It defines (respectively) a human readable name, the bundle identifier, a description, the OSGi specification, the version number of the bundle, the class name to invoke, a name of a package that will be made available to the outside world, the name of packages required from the outside world (dependencies).

Bundle Life-Cycle

The life-cycle of an OSGi bundle is the following:

1. INSTALLED – The bundle is successfully installed on the device within the OSGi context.
2. RESOLVED – All the Java classes required by the bundled are available.
3. STARTING – The `BundleActivator.start()` method is called on the activator class in the bundle.
4. ACTIVE – The bundled has been started successfully.
5. STOPPING – The `BundleActivator.stop()` method is called on the activator class in the bundle.
6. UNINSTALLED – The bundle has been uninstalled from the OSGi context definitively.

Service

A service in OSGi is defined by an interface and an implementation of that interface.

For example:

```
public interface DictionaryService {

    // Check for the existence of a word.
    public boolean checkWord(String word);

}
```

And:

```
public class DictionaryImpl implements DictionaryService {

    // The set of words contained in the dictionary.
    String[] m_dictionary = { "welcome", "to", "the", "osgi", "tutorial" };

    public boolean checkWord(String word) {

        word = word.toLowerCase();

        // This is very inefficient
        for (int i = 0; i < m_dictionary.length; i++) {
            if (m_dictionary[i].equals(word)) {
```

```
                return true;
            }
        }

        return false;

    }

}
```

Services objects are registered by the bundle as following:

```
public void start(BundleContext context) {

    Properties props = new Properties();
    props.put("Language", "English");

    context.registerService(DictionaryService.class.getName(),
        new DictionaryImpl(), props);

}
```

They are automatically unregistered when the bundle is stopped. Later, an instance of the service can be obtained as following from the bundle context:

```
public void serviceUser(BundleContext context) throws Exception {

    // Query for all dictionary service references matching any language.
    ServiceReference[] refs = context.getServiceReferences(
        DictionaryService.class.getName(), "(Language=*)");

    if (refs != null) {

        try {

            // Get a dictionary service
            DictionaryService dictionary =
                    (DictionaryService) context.getService(refs[0]);

            // Check a word
            if (dictionary.checkWord("MyWord")) {
                System.out.println("Correct.");
            } else {
                System.out.println("Incorrect.");
            }

            // Unget the dictionary service.
            context.ungetService(refs[0]);
```

```
        } catch (IOException ex) {

        }

    }

}
```

Publishing, Discovering and Binding

Conceptually, a *service* is registered in a *service provider* which *publishes* the corresponding *service description* in a *service registry*. A *service requester discovers* services from the service registry and *binds* to service providers to obtain them.

Within OSGi, all this happens within a bundle. It is responsible for all above operations.

Standard Services

OSGi offers several standard service with are accessible by bundle from interfaces their objects can implement. This includes a logging system centralizing log entries for example.

Limitations

The bundle model allows the definition of code dependencies through the manifest file. These are called *deployment dependencies*. However, it does not allow the definition of dependencies between services (i.e., *service dependencies*).

We know that the execution of JXTA services depend of each other (for example, the rendezvous service cannot run if the endpoint service is not available). This is a limitation.

Service Components

In OSGi, service components are similar to bundles, except that many service components can be deployed inside a bundle. Without entering into details, dependencies can be implemented between service components.

Overall Differences & Similarities

- JXTA & OSGI both support continuous deployment activities; modules and services in one case, bundles service components in the other.
- JXTA & OSGI both know Http transport.
- JXSE module life-cycle methods (`init()`, `startApp()`...) cannot be exchanged for/replaced with `BundleActivator` methods (`start()`, `stop()`...). The life-cycle of a JXTA/JXSE module has to be handled by OSGi Service Component dependencies.
- JXTA/JXSE allows user to add URLs to fetch Java classes implementing the `Module` interface from remote locations. This functionality is not available from OSGi.
- OSGi monitors changes in its services registry and notifies listeners with events (departure, arrivals or changes). This functionality is not available in JXTA/JXSE.

Feasibility

Scoping

One can identify three levels of integration between JXTA/JXSE and OSGi:

1. Running JXTA/JXSE as an OSGi service component
2. Rewrite JXTA/JXSE services as OSGi service components
3. Allow peers to load and run JXTA modules implemented as OSGi service components

JXTA/JXSE as an OSGi Service

Requirements

The current status of the code as of JXTA/JXSE release 2.5 would require a redesign of the WorldPeerGroup (WPG), the NetPeerGroup (NPG) and of the peergroup creation process. The WPG and NPG factories would (most probably) need to be refactored or replaced by a OSGi bundle where a peergroup factory would be made available.

Considering that the community has already expressed the need to review the creation process of peergroups, this might be an opportunity. Since OSGi requires proper API design, this would also be an opportunity to brush up a part of the current JXTA/JXSE API.

Some integration would have to be performed with the JXSE implementation of the JxtaLoader. A deterministic 'stopping' of JXTA/JXSE would be welcome too.

Assuming that the JXTA community chooses Apache Felix to integrate OSGi, the JXTA/JXSE service could be started using the delivered jxta.jar like an OSGi bundle by external applications. Else, in a standalone application, the OSGi functionality would be started by creating an instance of the `Felix` Java Class, from which the JXTA/JXSE component service would be started.

JXSE Modules as OSGi Service Components

Requirements

Rewriting delivered JXTA/JXSE modules into OSGi service components requires (at least) brushing up the API for standard and core JXTA services. This cannot be performed without performing the work required to make JXTA/JXSE available as a component service.

Achieving this level of integration would simplify the complexity of the code and of interactions between JXTA services. Much of the start-up and stop procedures could be delegated to OSGi.

One could also imagine JXTA/JXSE 'internal' services for cache management or thread management for example. These (together with other services) implementations could easily be exchanged as needed.

Ultimately, this might also open the possibility of running multiple peers within the same JVM, which would considerably facilitate the writing of QA tests for continuous integration.

Loading JXSE User Modules as OSGi Service Components

Requirements

This functionality would require providing an access to the delivered OSGi service provider in JXTA/JXSE. Some API would have to be created, upon which users could develop their own JXTA/JXSE services and module.

The fact that the JXSE JxtaLoader implementations knows about URLs to load remote code – and why not OSGi bundles – could motivate OSGi users to switch to JXTA/JXSE as there preferred communication and networking layer. The JXSE JxtaLoader also offers the Java security policiy layer via the `SecureClassLoader`.

A clarification is to be made regarding the relationship between a JXTA module/service and the fact that it belongs to a peer or to a peergroup.

Conclusions

There are synergies and mutual needs JXTA/JXSE and OSGi can fulfill for each other. A progressive integration of OSGi within JXTA/JXSE is possible and even advisable. The overall architecture and code of JXTA/JXSE would be simplified, together with its testing and continuous integration of new contributions.

The full integration of OSGi with JXTA/JXSE would provide a lot of functional value to both OSGi users and JXTA/JXSE users.

This would also be a good opportunity to clarify parts of the JXTA specifications and to give up the JXTA concept of 'application' within JXSE. JXTA/JXSE is not implemented as a standalone application, but rather as a service for applications.