THE NARADABROKERING USER'S GUIDE

USER'S GUIDE **VERSION 3.3.0**

COMMUNITY GRIDS LAB,
INDIANA UNIVERSITY
501 N. MORTON ST, SUITE 224
BLOOMINGTON IN 47404

http://www.naradabrokering.org

TABLE OF CONTENTS

1 Getting Started with NaradaBrokering ... 6
1.1 Basics ... 6
1.1.1 Requirements ... 6
1.1.2 Downloading additional jar files ... 7
1.2 Compiling the NaradaBrokering code base ... 7
1.2.1 Using Apache ANT ... 7
1.2.2 Compiling using the javac command ... 8
1.2.3 Checking the version of NaradaBrokering ... 8
1.2.4 Library Dependencies: ... 8
1.3 Conventions used in this manual ... 9

2 Configuring the broker ... 10
2.1 Configuring the ports for communications ... 10
2.2 Configuring a broker as stand-alone or part of a distributed network ... 11
2.3 Starting the Broker ... 11
2.3.1 For Windows ... 12
2.3.2 For UNIX environments ... 12

3 Setting up a distributed broker network ... 14
3.1 Requesting a node address: ... 15
3.1.1 Issuing a node address request ... 15
3.1.2 When Node Address Requests Fail ... 17
3.1.3 Which node assigns the node address for a given node? ... 17
3.2 Creating a gateway between broker nodes in a distributed network ... 17

4 Graphical deployment of Broker Networks ... 19
4.1 Compiling the Management framework ... 19
4.2 Terminology for the machines involved ... 19
4.3 The configuration files ... 19
4.4 Changes to the .bin files before running programs ... 20
4.5 VNC Servers ... 20
4.6 Preliminary setup ... 22
4.7 The GUI for deploying Broker Networks ... 24
4.7.1 In case of initialization problems ... 24
4.7.2 Main Window of the Deployment Panel ... 25
4.7.3 Resource Properties ... 26
4.7.4 Policies ... 27
4.7.5 Generating Topologies ... 29
4.7.6 Ring Topology ... 30
4.7.7 Cluster topology ... 31
4.7.8 Editing Links ... 33

4.7.9 Manually Creating Links ... 34
4.7.10 Shutting down the Broker Network ... 35

5 Specifying the creation of Links ... 36

5.1 Creating a link ... 36
5.2 A Code Snippet Detailing Link Creation ... 38
5.3 Instructions for SSL/HTTPS connections through a proxy ... 39
5.4 Using IPSec ... 40
5.4.1 IPSec Server: ... 40
5.4.2 IPSec Client: ... 42
5.4.3 Setting up of the IPSec Tunnel from NaradaBrokering clients ... 44

6 Developing NaradaBrokering Applications ... 46

6.1 Primer on events, synopsis, profiles and templates ... 46
6.2 Writing a simple NaradaBrokering client ... 47
6.2.1 Initializing the Client Service ... 47
6.2.2 Initializing communications with the broker ... 48
6.2.3 Initializing the consumer role ... 48
6.2.4 Initializing the producer role ... 50
6.2.5 Event Properties ... 51
6.3 Harnessing the available Qualities of Services ... 52
6.3.1 Consumer Constraints ... 52
6.3.2 Producer Constraints ... 52
6.3.3 Compression and Decompression Services ... 53
6.3.4 Reliable Delivery Services ... 54
6.3.4.1 Initializing the consumer ... 54
6.3.4.2 Initializing the producer ... 55
6.3.5 Managing Replays ... 56
6.3.5.1 Creating the appropriate `ReplayRequest` ... 56
6.3.5.2 Initiating Replays ... 57
6.3.6 Fragmentation and Coalescing ... 58
6.3.6.1 The Fragmentation Service ... 58
6.3.6.2 The Coalescing Service ... 59

7 Setting up the Repository Node ... 61

7.1 Creating the Database and Tables (Windows and Linux) ... 61
7.2 Using the Robust Node ... 62
7.3 The Robust Subscribers and Publishers ... 74

8 Writing JMS applications ... 78

8.1 Creating a TopicConnectionFactory ... 78
8.2 Initializing the Topic Session and Topic ... 78
8.3 Creating a Subscriber ... 79
8.4 Message Types ... 80
8.5 Creating a Publisher ... 80

8.6 Running the sample JMS chat application 81
8.7 Unsubscribing Topics 81

9 Broker Discovery 82

9.1 Discovering Brokers 82
9.2 Using the Broker Discovery Helper 82

10 Topic Creation & Discovery 84

10.1 Topic Creation 84
10.1.1 Starting the Topic Discovery Node 84
10.1.2 Creating Topics 84
10.2 Topic Discovery 86
10.2.1 Discovering Topics 86

11 Root Provider 88

11.1 Using the Root Provider 88
11.2 Loading Certificates and Keys 90

12 Security Framework 91

12.1 Creating Security Tokens and securing topics 91
12.1.1 Starting the Key Management Center 91
12.2 Creating Secure Topics 91
12.3 Signed Security Token Retrieval 93
12.4 Secure Publishing of events 94
12.5 Receiving Secure Events 95

13 The C++ Bridge for NaradaBrokering 97

13.1 C++ Socket Client for Naradabrokering 97
13.1.1 Configuration 97
13.1.1.1 Broker Configuration 97
13.1.1.2 Compiling the C++ Client 98
13.1.2 Simple Chat Client 98
13.1.3 The Architecture 98
13.1.4 Issues specific to Endianness 100
13.1.5 Simple Pub/Sub Example 100
13.2 C++ Bridge for NaradaBrokering (JNI-based) 100
13.2.1 Broker Configuration 101
13.2.2 Compiling the C++ Bridge 101
13.2.3 Simple Chat Client 102
13.2.4 The Architecture 102
13.2.5 How to Use the Bridge 102
13.2.6 Simple Publisher Example 104

14 Appendix A: Working with the codebase in IDEs 106

14.1 Incorporating the NaradaBrokering Codebase into Eclipse 106

14.1.1 Download NaradaBrokering and the Necessary Jars 106
14.1.2 Creating New Project Using Eclipse 106
14.1.3 Use NaradaBrokering in Your Project....112
14.2 Importing the codebase into JBuilder 114

15 Appendix B: The Broker Configuration File 116

1 Getting Started with NaradaBrokering

In this chapter we cover issues pertaining to getting a quick start on utilizing the NaradaBrokering System. The chapter covers issues pertaining to installing the software, compiling the code base and starting the individual brokers. The chapter also provides a discussion on setting up a distributed broker network.

1.1 Basics

The NaradaBrokering software is available for download at http://www.naradabrokering.org. The distribution is a zip file. When you unzip the file, the distribution is contained in a folder named **NaradaBrokering-x.y.z** where **x.y.z** corresponds to the version number of the NaradaBrokering release; here **x** indicates the major release and indicates a significant advancement in the software's capabilities; **y** indicates the minor release which adds incremental capabilities to the major release; finally, **z** indicates improvements to the minor release which is typically the result of bug fixes. The directory structure of a typical NaradaBrokering distribution is depicted in the figure below

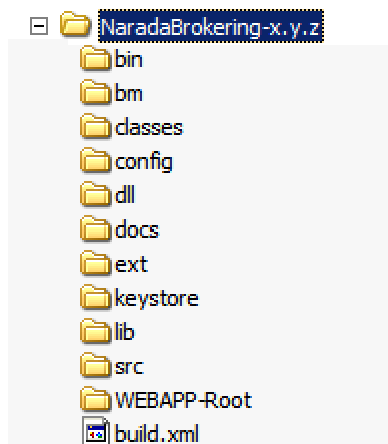


Figure 1: High level view of the NaradaBrokering distribution

1.1.1 Requirements

NaradaBrokering is written in Java and requires you to have **JRE/JDK 1.6** or higher. You can download the latest version of Java from http://java.sun.com . NaradaBrokering uses classes and features that are available in these newer versions of the Java Virtual Machine.

NaradaBrokering has been tested on Windows (NT/XP), Linux and Solaris based systems.

1.1.2 Downloading additional jar files

NaradaBrokering binaries are included in the distribution. However, you may need to download some of the necessary jar files without which you will have problems running the software effectively. Specifically, you need to download two pieces of software and include these jar files in your CLASSPATH.

JMS: The Java Message Service specification is a set of interfaces that abstract one-to-one and one-to-many communications between entities. You can download this jar file from http://java.sun.com/products/jms/index.html

JMF: The Java Media Framework is also needed to execute the multimedia related tools in NaradaBrokering. The latest version of JMF can be found at http://java.sun.com/products/java-media/jmf/

1.2 Compiling the NaradaBrokering code base

Start off by making sure that you are using **JDK-1.6 or higher** (you can verify this by typing **`java -version`**). Also check and see whether your **`javac`** is JDK-1.6.1 or higher by checking the path variable in case you have multiple Java SDK installations on your machine.

1.2.1 Using Apache ANT

The easiest way to compile the entire code base in the NaradaBrokering distribution is to use ANT. ANT is Java based build tool from Apache and can be downloaded from http://ant.apache.org/. We have included an XML file build.xml in the distribution which can be used to compile the entire NaradaBrokering source tree. Next, you also need to update the locations of the jms.jar and jmf.jar files specified in the build.xml file.

Once you have downloaded the ANT software and updated the locations of the jms.jar and jmf.jar files in the build.xml file, you can supply this **`build.xml`** file as a parameter to the ant command to compile the entire distribution. Thus, the command ant will rebuild the distribution. This rebuilding will generate a new **`NaradaBrokering.jar`** in the **`NB_HOME/lib`** directory, where **`NB_HOME`** corresponds to the location of the NaradaBrokering distribution on your machine.

1.2.2 Compiling using the javac command

If you aren't using ANT and you are trying to compile the sources from the command line you need to make sure that you did include the jar files included in the **lib** directory of the distribution in the CLASSPATH that you specified while compiling.

1.2.3 Checking the version of NaradaBrokering

There are many instances where developers have multiple instances of NaradaBrokering running. Its also conceivable that the jar files in your CLASSPATH may have multiple NaradaBrokering.jar files. To verify the version of NaradaBrokering that you are running please use the following command **java cgl.narada.util.Version**.

1.2.4 Library Dependencies:

Included below is the list of jar files needed for executing and accessing the entire NaradaBrokering functionality in addition to jms.jar and jmf.jar. The jar files listed below are currently being included in the distribution and are covered by their individual licenses.

Table 1: Summary of library dependencies

Software	Function	Availability	NB Distribution location
Xerces	XML parser	http://www.apache.org	lib/Xerces.jar
Xalan	XPath parser	http://www.apache.org	lib/Xalan.jar
ExoLab JMS Selector	SQL selector in openJMS	http://www.exolab.org	lib/exolabJMSselector.jar
ANTLR	Grammar functionality used by ExolabJMS selector mechanism	http://www.antlr.org	lib/antlr.jar
JXTA	P2P functionality	http://www.jxta.org	lib/jxta.org
Log4j	Logging facility	http://www.apache.org	lib/log4j-1.2.8.jar
MD5 library	Digest authentication which requires the MD5	Timothy W Macinta http://www.twmacinta.com	lib/MD5.jar Package: com.twmacinta.util.*
Base 64 Encoder/ Decoder Library	Needed for Basic User Authentication.	Robert Harder http://iharder.sourceforge.net/base64/	src/cgl/narada/util/ Base64.java
Digest	Needed for supporting	Clarke County Code	lib/

Authentication	Digest Authentication	Brewing Company. http://www.geocities.com/ballarke/Projects/HttpClient/	DigestAuthen.jar Package: digestauthe.*
Cryptix Cryptographic extensions	Used for implementing cryptographic functions	http://www.cryptix.org/	Under lib cryptix_jce-provider.jar cryptix_jce–compact.jar cryptix_jce–tests.jar cryptix_jce–api.jar

1.3 Conventions used in this manual

This is the font <Courier New, 11 pt, bold> used for Variable names and Class names

This is the font <Courier New, 11 pt, bold, red> used for specifying executables & directory locations

This is the font <Courier New, 11pt> used for Code Snippets

This is the font <Verdana, 10 pt> used for everything else

2 Configuring the broker

Included in the distribution is a file for configuring the broker. This file (NB_HOME/config/ BrokerConfiguration.txt) could be used for configuring the network communication ports for a broker and for other properties that control the broker's behavior. This configuration file is included in the appendix of this manual.

2.1 Configuring the ports for communications

A broker in NaradaBrokering can communicate over multiple ports over different transport protocols. The protocols supported within NaradaBrokering include TCP, UDP, Multicast, SSL, HTTP and RTP. The TCP communications include support for both blocking and non-blocking IO. Included below is a table outlining the parameters, default values for them and their accompanying functions.

Table 2: List of ports that are used for communications by specific transports

PARAMETER	DEFAULT	FUNCTIONALITY
NIOTCPBrokerPort	3045	This parameter specifies the port number for non-blocking TCP communications with the broker.
UDPBrokerPort	3045	This parameter specifies the port at which the broker listens to for datagram communications. This port is ideal for transient communications with the broker.
MulticastGroupHost/ MulticastGroupPort	224.224.224.224/ 4045	This pertains to communicating with the broker using multicast. The port specified here has to be different from the one specified for the UDPBrokerPort.
TCPBrokerPort	5045	This parameter specifies the port number for blocking TCP-based communications with the broker
PoolTCPBrokerPort	6045	This is an experimental part of the NaradaBrokering system which concerns the use of thread pool to manage concurrent connections. This feature eliminates the need to have a thread associated with every connection.

In addition to this NaradaBrokering can also communicate using SSL over port 443 and HTTP over port 80. NaradaBrokering now incorporates support for IPSec. To use this particular feature, one does not need to any configure specific ports which the broker would use to accept connections or incoming traffic. Once the tunnel has been set up, all registered transports can use the tunnel for communications with the corresponding ports (as listed in the Table 2).

2.2 Configuring a broker as stand-alone or part of a distributed network

Every broker in NaradaBrokering has an ID associated with it. This address is assigned depending on how the broker is configured for use. A broker that intends to be part of a distributed network needs to retrieve its address by issuing a request to one of the brokers (with an assigned address) within the distributed network. If, however, a broker is being run in the stand-alone mode the broker assigns itself a default address.

The parameter **AssignedAddress** controls this behavior. If this is set to **true** the broker assigns itself a default address and begins operation in stand-alone mode. Other brokers can contact this broker to help set up the distributed network. Please note that the first broker on a NB broker network assigns its own address.

If the **AssignedAddress** parameter is set to **false** the broker does not assign itself an address and is ready to be part of a distributed broker network.

2.3 Starting the Broker

In the **bin** directory of the NaradaBrokering installation please update the **NB_HOME** variable in the **.bat** (and .**sh**) executable scripts. The **NB_HOME** variable points to the location of the NaradaBrokering installation. For example the **NB_HOME** variable could be `/home/users/smith/NaradaBrokering-3.2.0`. Note that that the location of the installation directory does not have a trailing slash "/".

Table 3: The startbr.sh file that is used for starting the broker process

```
export NB_HOME=..
brokerConfigFile=${NB_HOME}/config/BrokerConfiguration.txt
serviceConfigFile=${NB_HOME}/config/ServiceConfiguration.txt
brokerCommunicatorPort=11111
brokerCommunicatorFile=${NB_HOME}/config/uuid.txt

cp=.

for i in ${NB_HOME}/lib/*.jar;
```

```
 do cp=$i:${cp}
done

for i in ${NB_HOME}/lib/*.zip;
  do cp=$i:${cp}
done

java -Xmx260m -Xms260m -Xmn32m -XX:SurvivorRatio=10 -classpath $cp
cgl.narada.node.BrokerNode --brokerConfig=$brokerConfigFile --
serviceConfig=$serviceConfigFile --
brokerConnmmunicatorPort=$brokerCommunicatorPort --
brokerCommunicatorFile=$brokerCommunicatorFile&
```

There are two configuration files that the broker uses. The first file is related to Broker configurations (such as port numbers etc) while the second is related services loaded by the broker.

In addition to this, there is a third parameter – the broker communicator port. This feature was introduced to allow the broker to be run as a background process while retaining the ability to interactively issue commands to the broker process.

If you need to start multiple brokers on the same machine, you will need to update your broker communication ports in both the **startbroker** (sh and bat files) and **brokerInteract** (sh and bat files) appropriately.

2.3.1 For Windows

For Microsoft OS users the file that needs to be updated is the startBroker.bat file. To start broker under Windows you can simply double click the **startBroker** icon. For Windows-NT please also include the **%NB_HOME%\dll** in your path variable. This is needed to enable automatic detection of proxy settings using the WinINET API. This is useful during communication through proxies and firewalls.

2.3.2 For UNIX environments

For UNIX users the file to modify is the startbr.sh file. The first time you try to execute this file you would also need to make the file executable by using the command **chmod +x startbr.sh**. To start the broker under Linux/Unix use the following command in the $NB_HOME/bin directory – **./startbr.sh**

Within the UNIX environment we have included another file (**stopbr.sh**) which allows one to shutdown a currently running broker process using the command **`./startbr.sh`**.

3 Setting up a distributed broker network

In this chapter we describe the setting up of a distributed broker network. But before we do that we digress on the overlay structure that NaradaBrokering imposes on the distributed broker network

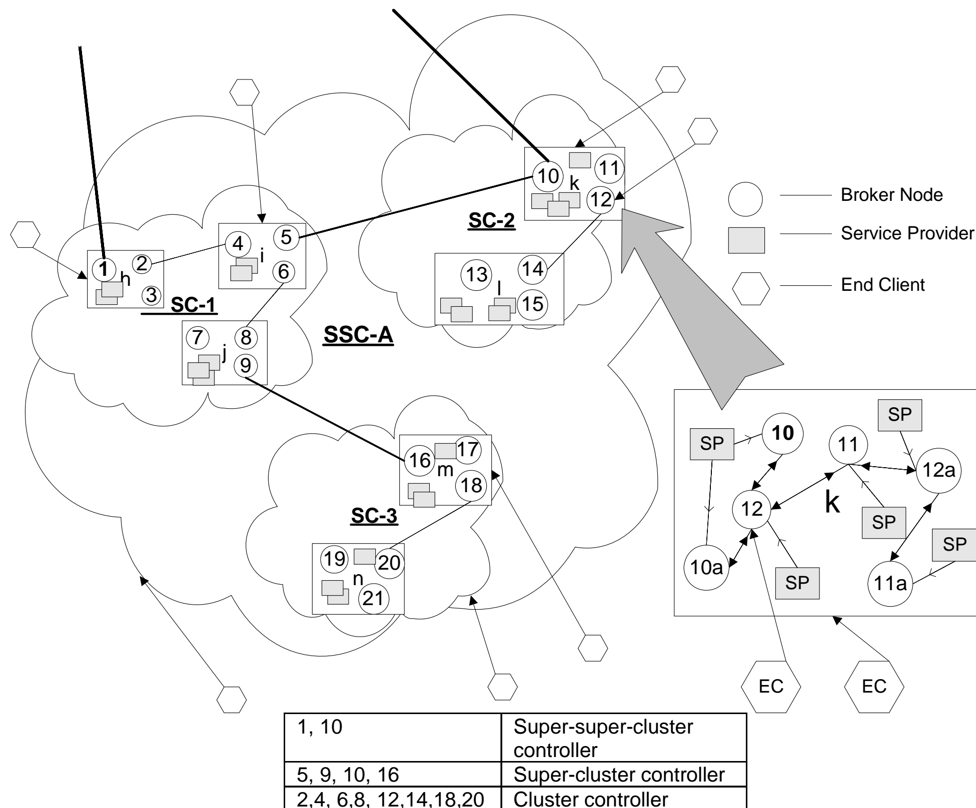


Figure 2:An example of a NaradaBrokering broker network sub-section

In NaradaBrokering we impose a hierarchical structure on the broker network, where a broker is part of a cluster that is part of a super-cluster, which in turn is part of a super-super-cluster and so on. Figure 1 depicts a sub-system comprising of a super-super-cluster **SSC-A** with 3 super-clusters **SC-1**, **SC-2** and **SC-3** each of which have clusters that in turn are comprised of broker nodes. Clusters comprise strongly connected brokers with multiple links to brokers in other clusters, ensuring alternate communication routes during failures. Within every unit (cluster, super-cluster and so on), there is at least one unit-controller, which provides a gateway to nodes in other units. For example in figure 1, cluster controller node **20** provides a gateway to nodes in cluster **m**. Creation of broker network maps (BNMs) and the detection of network partitions are easily achieved in this topology.

Please note that in NaradaBrokering we limit the number of units within a super-unit to 32. Thus, there can be only 32 brokers within a cluster. Similarly, there can only be 32 clusters within a super-cluster and so on. Figure 2 depicts the NaradaBrokering ID associated with a broker node. The NaradaBrokering broker addresses are of the form **23.20.31.14** – where **14** corresponds to the broker id within the cluster **31** of super-cluster **20** within the super-super-cluster **23**.

The clusters in the overlay structure may or may not correspond to actual clusters. Sometimes a cluster may comprise of broker processes running on geographically closer machines. Ideally, brokers within a cluster would comprise machines which can route messages very efficiently between each other. Also brokers within a cluster will have multiple links between them to ensure alternate communication paths during broker failures.

Establishing a NaradaBrokering connection between 2 brokers is different from simply establishing a socket connection between them. Establishing a socket connection between broker nodes is simply a precursor to issuing requests to set up a broker node within the broker network.

3.1 Requesting a node address:

As mentioned earlier, setting up of distributed broker network requires that the first broker within the network has a self assigned address. This self-assigned default address for the starting node is **1.1.1.1**. When broker nodes are being added to the system, depending on their *node creation requests* (issued to brokers with assigned addresses) appropriate *logical units* are created within the system. A broker performs 3 steps to facilitate its addition into the distributed broker network. We enumerate these below

- Set the **`AssignedAddress`** to `false` in the broker configuration file.
- Connect to one of the brokers within the distributed broker network. A broker is part of a distributed broker network only if it has a unique NaradaBrokering address assigned to it.
- Next, the broker creates a request for setting up of this node within the broker network.

We will also include an example below will describe the process of adding broker nodes within the system.

3.1.1 Issuing a node address request

When the broker process is running it continues to accept command line inputs from the broker administrator. We are currently in the process of addition a GUI based version of broker administration to the broker process. This section concentrates on command line inputs for now. The command line inputs are specified using the **brokerInteract** (bat or

sh) file available in the **bin** directory. This was done so that users can continue to interact with the broker even though it is running in the background mode in Unix systems. Typing an "**h**" on the command line of this program lists the set of commands that can be issued.

The first step involves the creation of a socket connection to one of the nodes within the broker network. To do this the command that is issued is "**c <hostname> <port-number> <transport>**" where hostname corresponds to the IP addresses or hostname of the machine hosting the broker process. The port-number and transport correspond to the port over which the broker is listening to communication and the transport protocol that is used for communications over that port. Thus if a broker is listening to TCP communications over port 5045 the connection command would be "**c everest.ucs.indiana.edu 5045 t**".

The process of creating a connection returns a link-ID which snapshots information pertaining to the created connection. This ID is then used in the issuing of the node address request. The command for issuing a node address request is "**na <link-id> <address-level>**", where link-id corresponds to the connection ID mentioned earlier. The address-level can vary from zero to three (0-3) by default. An example usage is the following: "**na tcp://everest.ucs.indiana.edu:5045 0**". We now enumerate how the address level will relate to the organization of the broker network. Also for the purposes of discussion let us assume that broker node that the requesting-broker is interacting with has an address **2.5.7.9**

Address level	How the request translates within the system
0	This implies that the requesting broker seeks to be a part of the cluster that querying-broker is a part of. The address assigned to the requesting broker could be of the form **2.5.7.10**
1	This implies that the requesting broker seeks to create a new cluster within the super-cluster the querying-broker is a part of. The address assigned to the requesting broker could be of the form **2.5.8.1**. This newly created cluster contains only one broker node – the requesting broker.
2	This implies that the requesting broker seeks to create a new super-cluster within the super-super-cluster the querying-broker is a part of. The address assigned to the requesting broker could be of the form **2.6.1.1**. This newly created super-cluster contains only one broker node – the requesting broker.
3	This implies that the requesting broker seeks to create a new super-super-cluster within the brokering network the querying-broker is a part of. The address assigned to the requesting broker could be of the form **3.1.1.1**. This newly created super-super-cluster contains only one broker node – the requesting broker.

3.1.2 When Node Address Requests Fail

Node address requests can fail for one of three of reasons. First, if the number of sub-units within a unit has exceeded the maximum threshold of 32. Any request that implies the creation of an additional sub-unit within this unit will result in a failure. Thus if there are already 32 brokers within a cluster, a node address request with address-level=0 will result in a failure.

Second, a node address request will fail if the querying-broker has not been assigned a NaradaBrokering address. Finally, the process of assigning a node address can involve different nodes depending on the level of the request. Failures in intermediate brokers during this process can result in problems with assigning a node address to the requesting broker.

3.1.3 Which node assigns the node address for a given node?

The node set up request, if successful, assigns the broker requesting to be part of the network, a NaradaBrokering address. Depending on the node address request the address for the node is assigned by different nodes within the brokering network. If the broker seeks to be part of a cluster, the address is assigned by the lowest numbered broker within the cluster the broker would be a part of. If the broker issues a request with address-level=1 the address is assigned by the lowest numbered broker within the lowest numbered cluster in the super-cluster the broker seeks to be. The same pattern is followed for increasingly higher address levels.

3.2 Creating a gateway between broker nodes in a distributed network

Establishing a link to another broker is just a precursor to creating a connection that will be deployed for efficient routings within the system. We call this connection a *gateway* to distinguish it clearly from simple socket connections or simply establishing communication links. Depending on the gateway that is created between two nodes, they end up as unit controllers. For example if a gateway is established between brokers in different clusters (but within the same super-cluster) both these nodes will be designated as cluster-controllers within the system. Brokers can also set up gateways to other brokers within its cluster.

The first step to establishing a gateway that can be deployed for efficient disseminations is the creation of a link to that broker. This is similar to what we discussed in the earlier section. The command that is issued is "**`c <hostname> <port-number> <transport>`**" where hostname corresponds to the IP addresses or hostname of the machine hosting the

broker process. The process of creating a connection returns a link-ID which snapshots information pertaining to the created connection.

To create a gateway between brokers the request is of the form "**ga <link-id> <connection-level>**", where link-id corresponds to the established connection ID between the nodes. The connection-level provides an indication of the type of controller a node seeks to be.

There are certain rules that must be adhered to for the creation of gateways between two broker nodes. Brokers within a cluster can only establish gateways with each other that are of level 0. Brokers in two different clusters but within the same super-cluster can establish a gateway that can only be of level 1. Establishing such a gateway link also results in these endpoint nodes being designated as cluster controllers. The scheme works similarly for higher levels.

4 Graphical deployment of Broker Networks

In this section we describe the graphical deployment of broker networks. This software, HPSearch, is available for download from the NaradaBrokering project website. In this section we will provide detailed instructions on setting up broker networks graphically.

4.1 Compiling the Management framework

Currently the framework completely depends on NaradaBrokering for all its dependencies. Make sure you have the latest NaradaBrokering setup. Further, the installation is precompiled with Java 6. If you need to recompile for some reason, compilation is based on Apache Ant. To compile, issue the following command:

```
ant -DNB_HOME=path_to_nb_home jar
```

4.2 Terminology for the machines involved

For clarity of discussions we will be referring to two sets of machines. The first set of machines **B** = {B1, B2 .... }, the broker machines, will be the machines on which the brokers will run. The second set **M** typically will have only one machine where the core management components run, and from where you launch Graphical User Interfaces (GUI) to manage and deploy broker networks. A given management machine, can manage upto 700 brokers; since, one would typically not go beyond this we need to use only one machine. A discussion of creating a hierarchy of management nodes **M** = {M1, M2, ...} to manage extremely large broker networks is included at the end of this chapter.

4.3 The configuration files

Configuration files for the HPSearch system are available in the conf directory of the distribution. There are three configuration files: two of these are typically not modified. Modifications, if any, should be done on the management machine.

1. MGMT_HOME/conf/mgmtSystem.conf
   This file contains configuration information for the management framework components. The only change one would do is to replace "localhost" with the fully qualified name of the management machine, M, on which you decide to run the management components. Thus, use gf1.ucs.indiana.edu instead of gf1.
2. MGMT_HOME/conf/defaultMessagingNode.conf
   This file contains port information for a communications node used by the management framework. The ONLY reason to modify this file is if you feel that the default ports used by this component is unacceptable.
3. MGMT_HOME/conf/system.conf
   This file is also least likely to change and contains various timeouts, heartbeat intervals, retry counts.

If you plan on deploying a broker network with brokers running on machines {B1, B2, B3 ..}, you need to copy the modified mgmtSystem.conf file to the MGMT_HOME/conf/

directory of the machines { B1, B2, B3 ..}. If the machines involved {M, B1, B2, B3 ...} mount the same file-system, then you don't need to perform this step. It is still a good idea to make sure that all the broker machines {B1, B2, B3 ...} see the modified mgmtSystem.conf file with the fully qualified DNS name of the management machine.

4.4 Changes to the .bin files before running programs

You may need to make the following changes before running the management framework

1. In `MGMT_HOME/bin/setEnv.bat`(.sh), Set the value of **NB_HOME**
2. Change permissions on all the .sh files in the `MGMT_HOME/bin` directory to make sure that they have execute permissions. Executing the flowing command in the `MGMT_HOME/bin` directory ensures this: `chmod +x *.sh`

4.5 VNC Servers

If you are plan on running the management components on a machine (M1) that you connect to remotely (from a Windows machine), and if you are running Unix on this management machine, M1, you will need to use a VNC client. This is because the management machine will spawn two GUIs that it won't be able to spawn otherwise. To do this, you will first need to start a VNC Server on the management machine in question (e.g. gf8.ucs.indiana.edu)

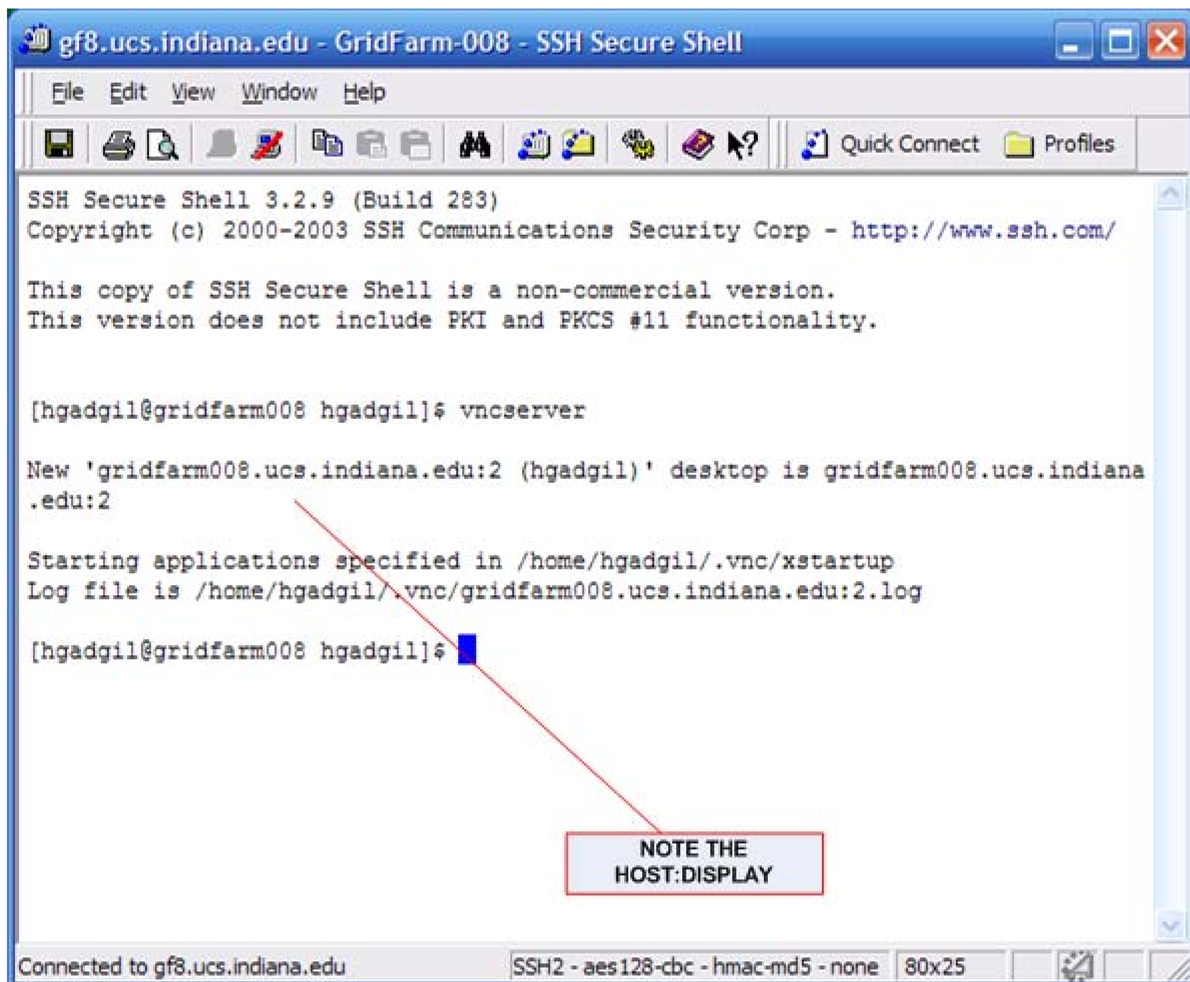


Figure 3: Starting a VNC Server

To connect to the VNC Server you will need a VNC Client (e.g. TightVNC). Once you do this you will be able to launch GUIs.

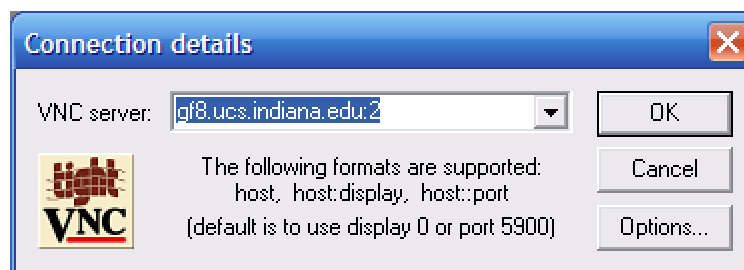


Figure 4: Using a VNC client to connect to the VNC Server

4.6 Preliminary setup

In this section we describe the preliminary steps that one needs to perform prior to being able to deploy broker networks graphically. These steps are order-sensitive, so please make sure that you do not perform these steps out-of-order. In each step we will also specify the machine on which a given program will be executed.

All the **.sh** files (or **.bat** files in the case of Windows) are being executed from the MGMT_HOME/bin directory. On Windows machines, to execute a .bat file you simply double-click the file in question. The remainder of this section will specify instructions for Unix based systems.

Step 1: Running the Fork daemons
The fork daemon needs to be running on ALL machines: the broker machines {B1, B2, ...} and also the management machine {M1}

./runForkDaemon.sh -- executeInTerminal

It is a good idea to use the executeInTerminal parameter if you are doing this for the first time since it simplifies the debugging process in case there are problems. If you do decide to submit it as a background job, at a later time, the logging output goes to **MGMT_HOME/logs/PROCESS.log** file.

The Default port used by the fork daemon is 65535. This can be changed while executing the fork daemon by specifying an additional parameter (-- port) to specify another port.

Step 2: Run the bootstrap node on the management machine
If you are accessing the management machine (M1) remotely from a Windows machine, we assume that you have performed steps to ensure that the GUI can be launched: one way of doing this was outlined in the preceding section.

To run the bootstrap node on the management machine type the following command in the MGMT_HOME/bin directory: **./bootstrapUI**. This will launch the Bootstrap Management Console which is depicted in Figure 5.

Clicking on the **Refresh** button, reloads the status of the bootstrap node currently being shown in "*Location of the ROOT Node Web Service*". If this service is unreachable, then the **Instantiate** button is activated which can be clicked to start the configured ROOT Bootstrap node by sending a message to the ROOT Node Fork Process Locator.

Clicking the **Instantiate** button causes a few .sh scripts to execute in different terminals that get launched through the GUI. These include: runBootStrapService, startRegistry, startMessagingNode and startManagerWithHealthCheck.

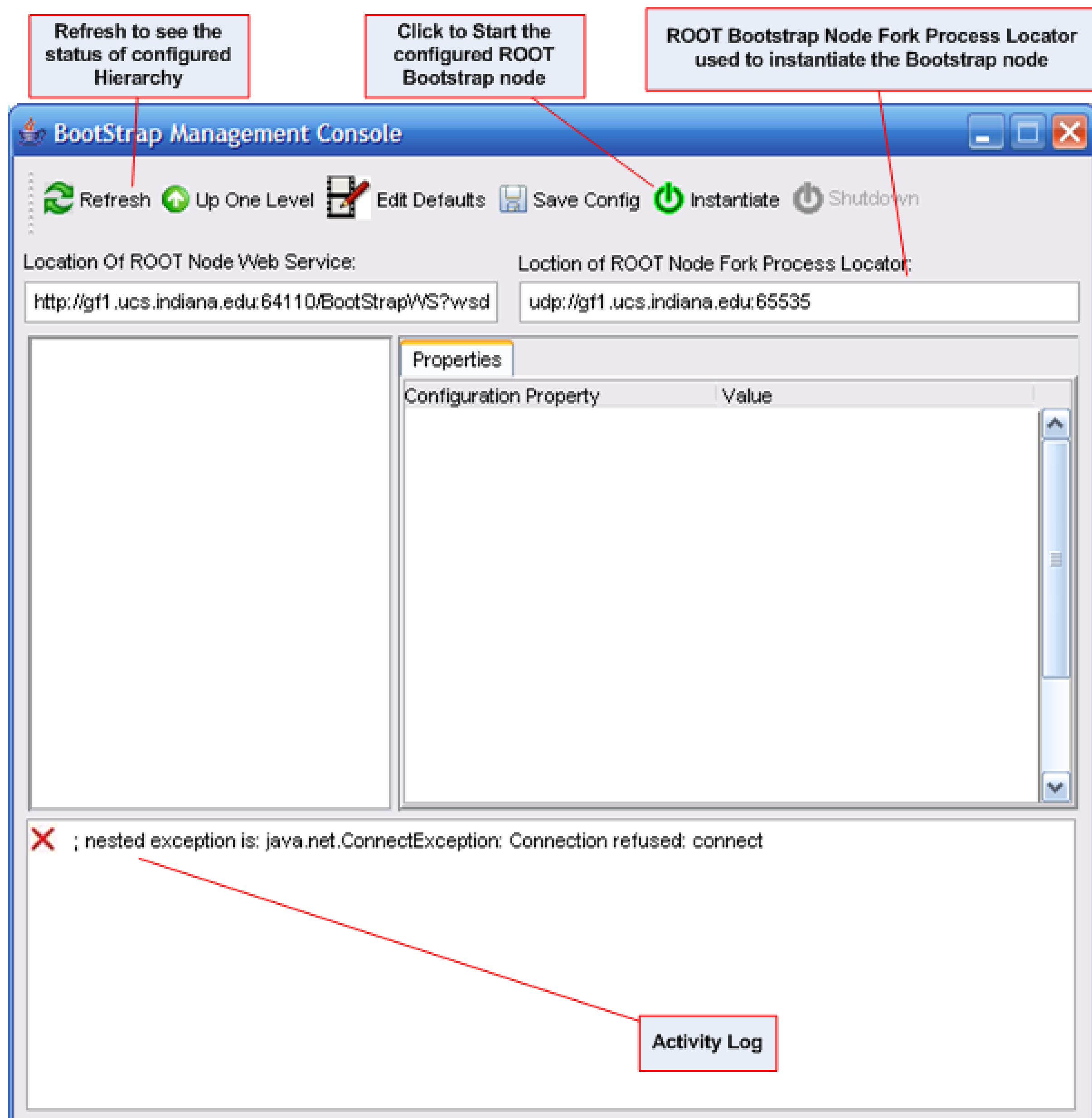


Figure 5: The Bootstrap Management Console

Step 3: Start the Broker Service Adaptors on the broker machines {B1, B2 …}
You then need to start the broker service adaptors (BSA) on the all the machines {B1, B2 …} where you intend to deploy the brokers. To do so execute the following command in the `MGMT_HOME/bin` directory of all the broker machines {B1, B2, …} :
`./runBrokerServiceAdapter.sh`

Step 4: Do a Refresh on the Bootstrap Console
Doing a Refresh on the Bootstrap Console will now launch the `startManager` in addition to the ones that were spawned-off in Step 2.

4.7 The GUI for deploying Broker Networks

In this section, we focus on the Broker Management GUI which is used to deploy broker networks. This GUI will be launched on the management machine (M1).

To launch this GUI, on the management machine (M1), you need to type the following command in the `MGMT_HOME/bin` directory: **`./userUI.sh`**

4.7.1 In case of initialization problems

If the system is NOT properly configured OR if the configured bootstrap node cannot be located after several retries the system does error reporting. In the error reports mode, a dialog box will pop-up prompting for a different location of the bootstrap service as shown in Figure 6.

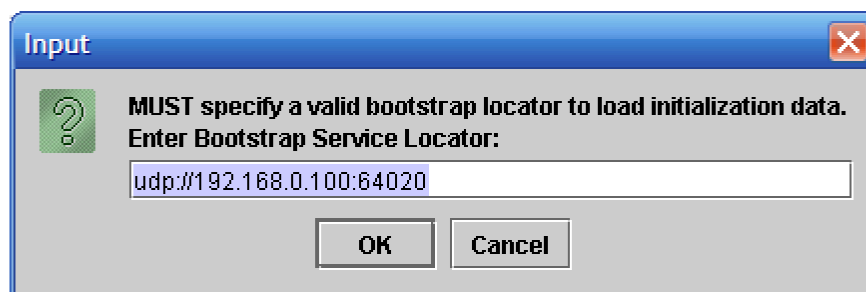


Figure 6: Prompt for Bootstrap Service Locator

Finally, if the bootstrap node cannot be contacted after several attempts a confirmation dialog (depicted in Figure 7) will be seen by the user.

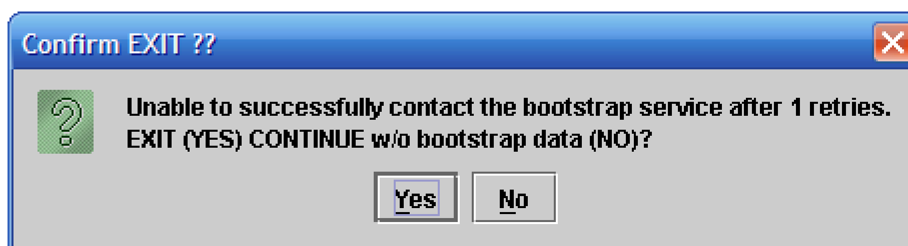


Figure 7: Dialog box if a Bootstrap node could not be located

On the contrary, if the bootstrap node was indeed correctly contacted, then the main window of the deployment console is shown

4.7.2 Main Window of the Deployment Panel

The main window (Figure 8) shows a list of available the Broker Service Adapter nodes and their associated registries. By selecting a node from the tree (left pane), one can view / set properties specific to the selected resource. In the left-pane, one needs to scroll to the right to see the complete IP address of the machines where the broker service adaptors have been started. The right pane consists of various resource specific tabs for configuring the selected resource.

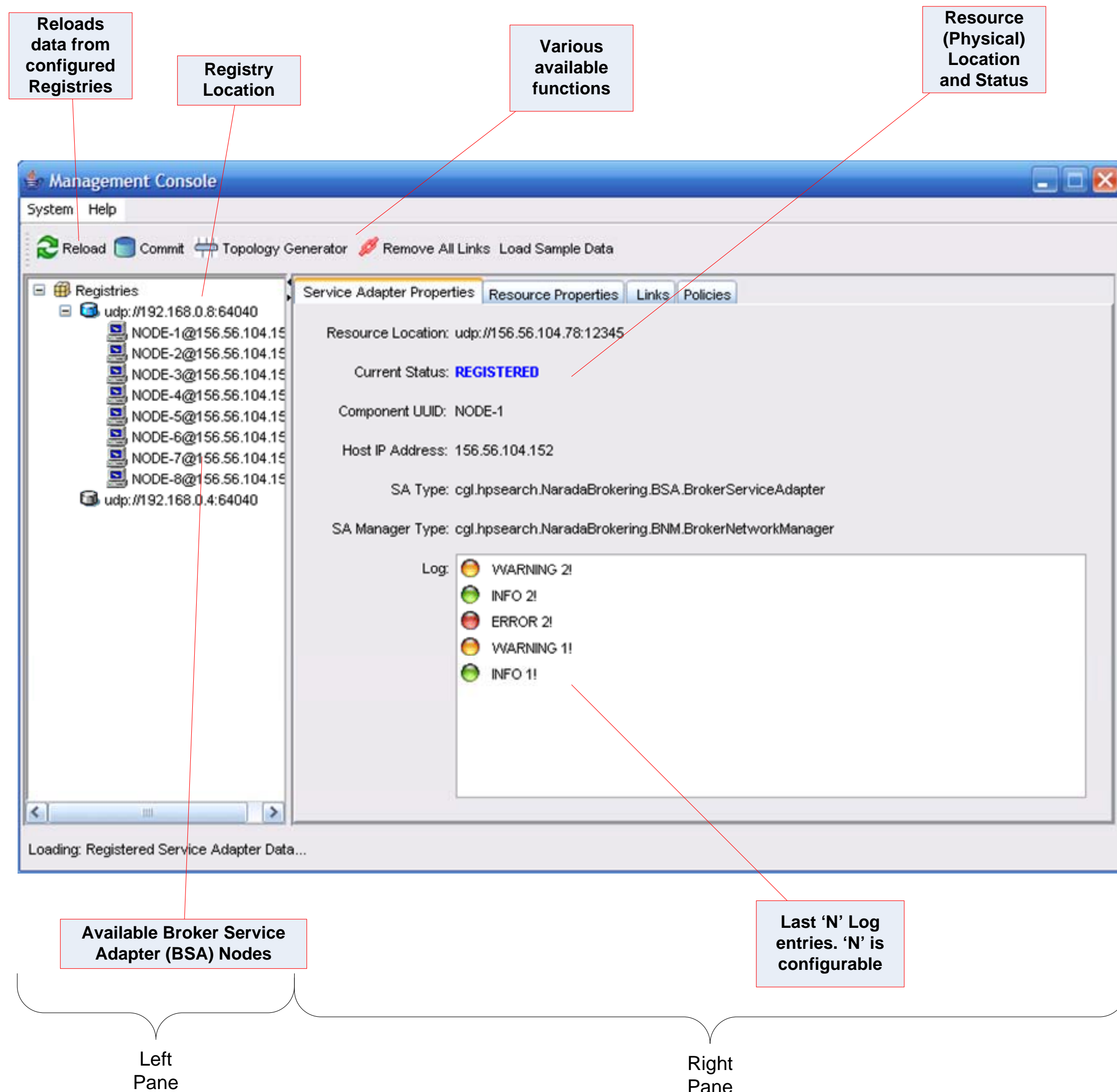


Figure 8: Main Window of the deployment console

The **Reload** button on the toolbar, reloads data from the registry. This overwrites the current user state and configuration.
Commit button is used to save all changes to the registry.

The **Topology Generator** button starts the topology generator which uses default topology generation algorithms. If a user-specific topology is desired, then the **Links** tab can be used to create and deploy a user-defined topology. Currently the Topology Generator provides support for 2 topologies *RING* and *CLUSTER*.
Remove All Links deletes all current links from the registry after the next commit.
The **Load Sample Data** is for debugging purposes to check the User interface functionality.
We now discuss the various tabs and functionalities of the GUI. The functionality depicted here is very specific to Broker Management.

4.7.3 Resource Properties

Figure 9 shows the main Resource properties window.

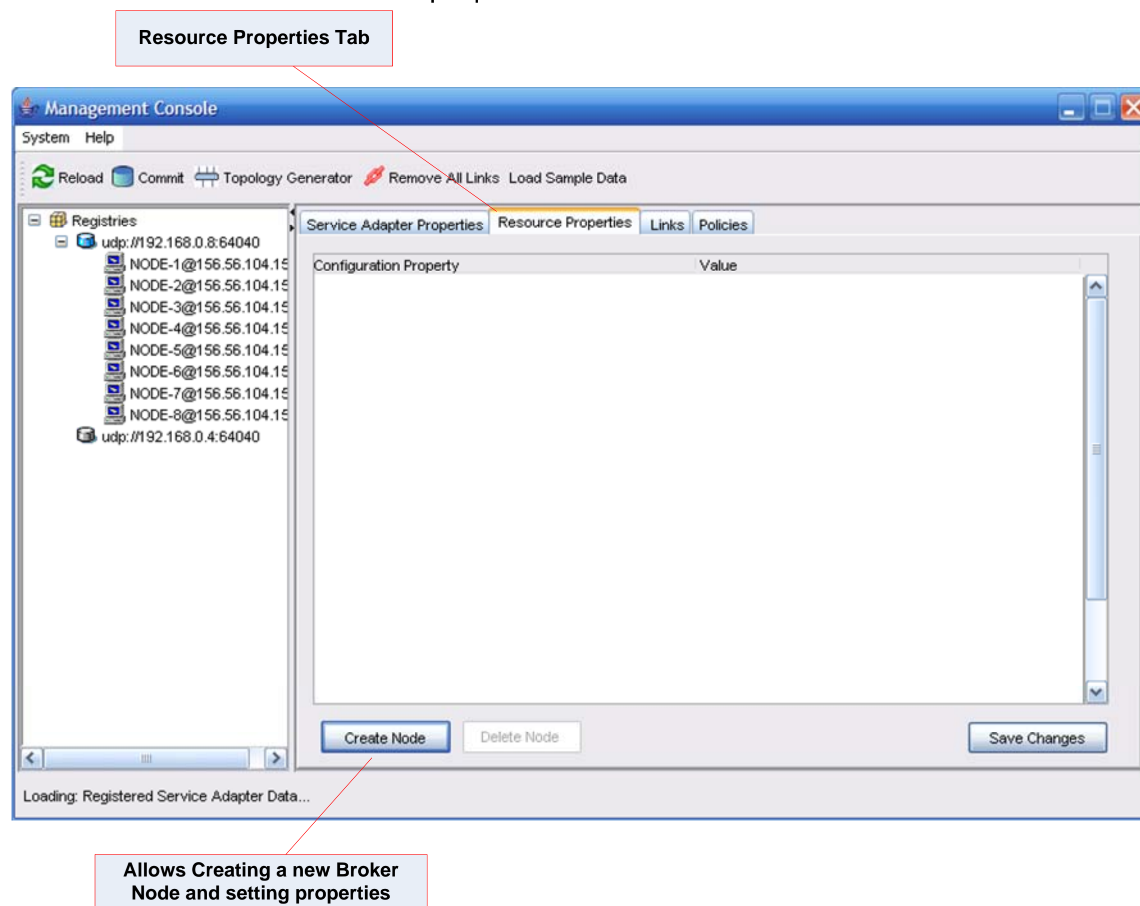


Figure 9: Resource Properties

The resource properties tab shows an editable table of *Configuration Properties* and their *Values*. Currently new values cannot be created, however existing values can be edited. This allows a user to configure a broker node to run specific services (such as, Run TCP and UDP transport on specified ports but do not run HTTPS/SSL and HTTP etc...).

The first step is to create a new node and change the default values if needed. This can be done by clicking the **Create Node** button. Figure 10 shows the default configuration properties after creating a new node.

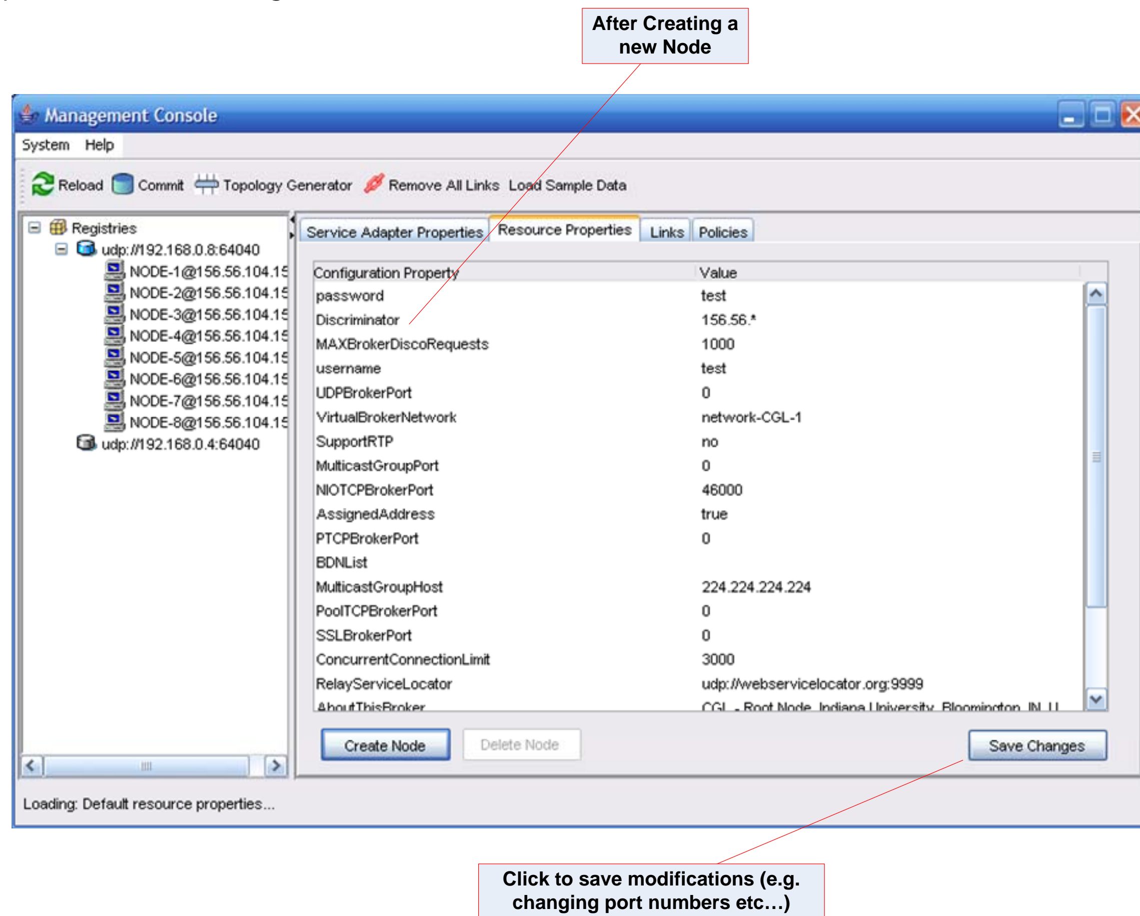


Figure 10: Default properties after creating a new node

To make any changes, simply double click the *"Value"* and press *"Enter"* when done.

Finally, the changes to a node's configuration may be saved (on the user's side) by clicking **Save Changes**. You then also need to **Commit** to ensure that these changes are registered. Failure to **Commit** will simply result in creation/changes to be discarded.

4.7.4 Policies

Failure of nodes would cause the application using the broker to function erratically. The usual method is to re-instantiate a new broker process manually. Whenever possible, this may be automated by setting the appropriate policy. The default policy, depicted in Figure 11, is to wait for *"User Interaction"* which simply put, *"Does Nothing"*.

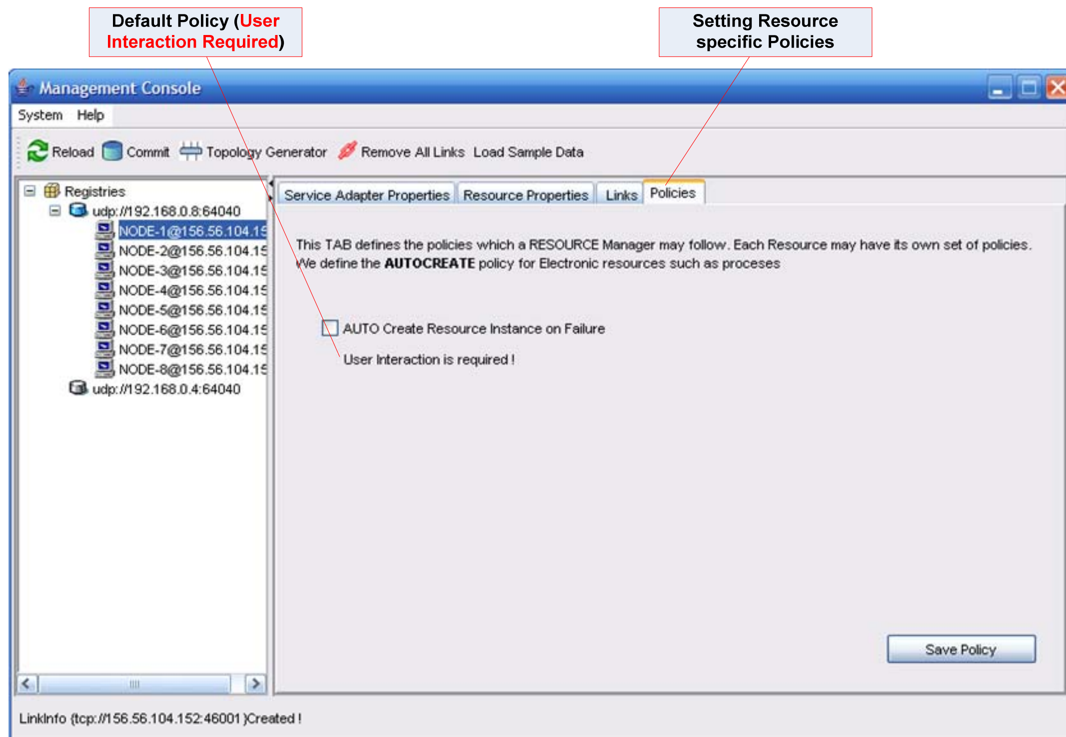


Figure 11: The default policy (Require User Interaction)

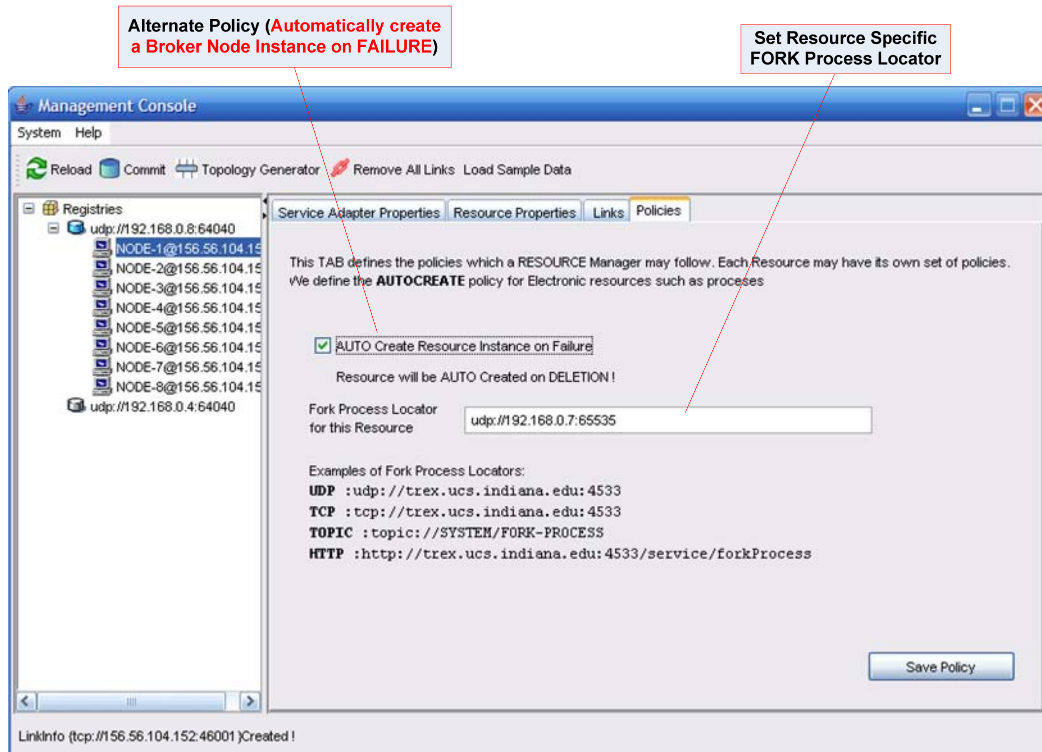


Figure 12: Alternate Policy (Automatically spawn a new Broker)

An alternate policy, depicted in Figure 12, is to use one of the *Fork Process Daemons* to spawn a broker process and use the newly spawned process in lieu of the failed broker process. The following **MUST** be noted for using this feature.

- In the current prototype implementation, **Only** *Fork Process Daemons* directly accessible (via UDP / TCP /HTTP or via a NB topic) can be used to spawn a new process.
- A failed process is typically indistinguishable from an extremely slow one. The determination of a process failure is solely dependent on missing heartbeats and the inability of the manager to successfully establish a contact with the target resource after several retries.

4.7.5 Generating Topologies

The **Topology Generator** button on the toolbar starts the topology generator module. Currently we have implemented a *RING* and a *CLUSTER* topology generator. Each of these topologies has specific characteristics. The main window for the topology generator is shown in Figure 13.

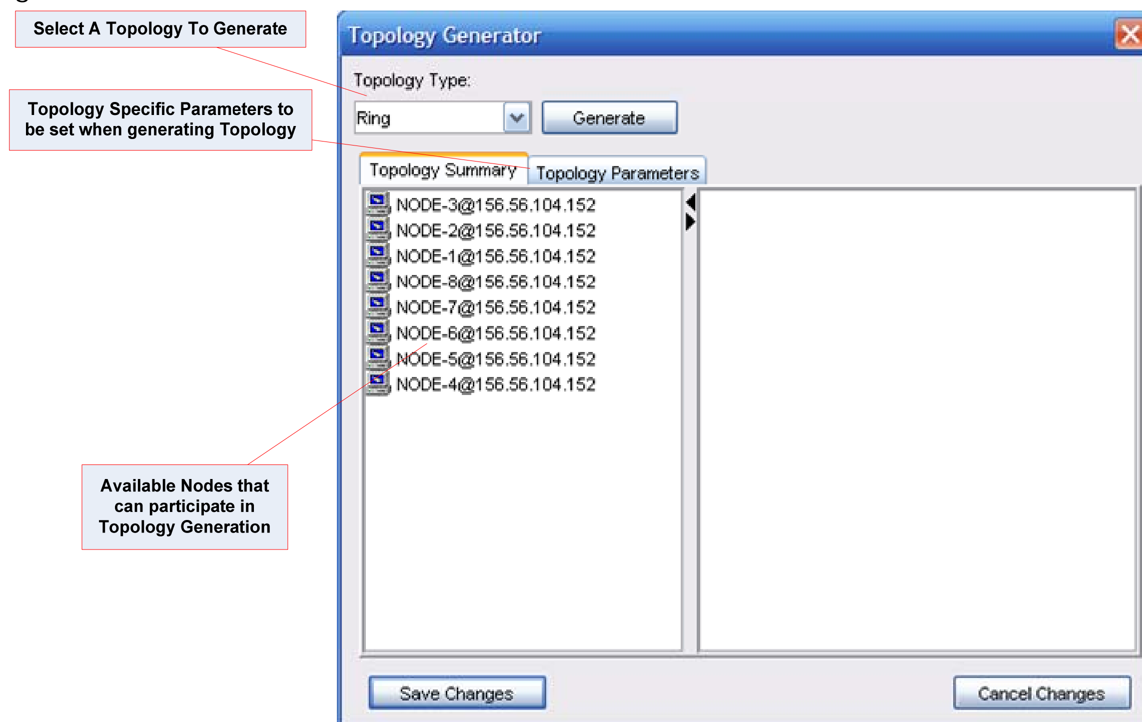


Figure 13: Main window for the Topology Generator

On the left side is a list of available nodes. An *Available Node* is defined as a node which was *created* using the **Create Node** on the *Resource Properties* page, and then *and committed* using **Commit**. Such a node is assumed to be completely configured and any

changes to this node after the links generation process would result in an incorrect deployment of the broker topology.

Figure 14: A warning dialog prompting for the confirmation of link deletion

The type of topology to generate can be selected from the drop-down list, and after setting topology specific parameters on the **Topology Parameters** tab, the user clicks **Generate, Save Changes,** and then **Commit** to generate the topology. The topology generation deletes all previous links and creates new links. A warning is issued (as shown in **Error! Reference source not found.**) before the topology generation is started. We now show the *RING* and *CLUSTER* topology generation on a sample data set.

4.7.6 Ring Topology

The Ring topology does not have any major topology specific parameters. When deploying broker network involving brokers behind NAT devices, a third party relay server (present in a non-NATed network) is used. The server location is configured as shown in Figure 15.

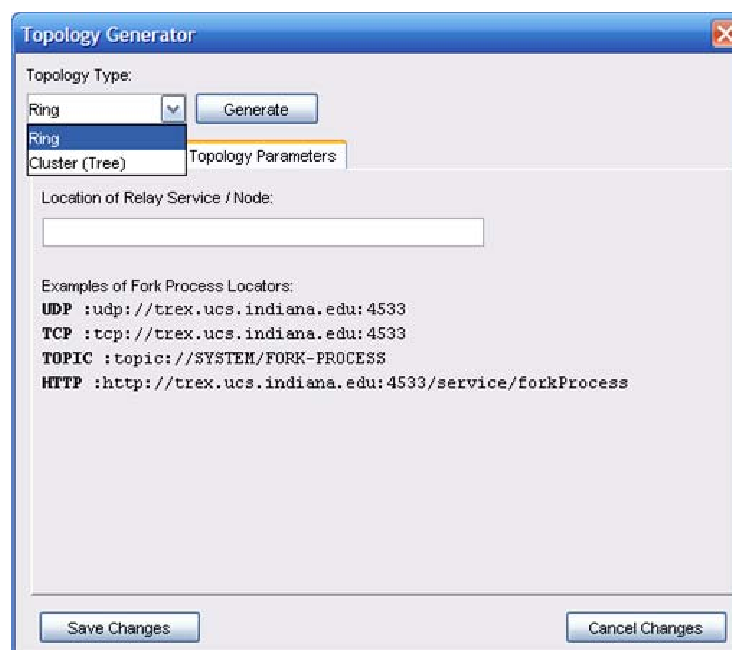


Figure 15: Ring topology parameters

When **Generate** is clicked the output for an 8-node network is shown in Figure 16. To complete the generation of the Topology and the linking-up of the nodes click **Save Changes**, and then **Commit** to generate the topology.

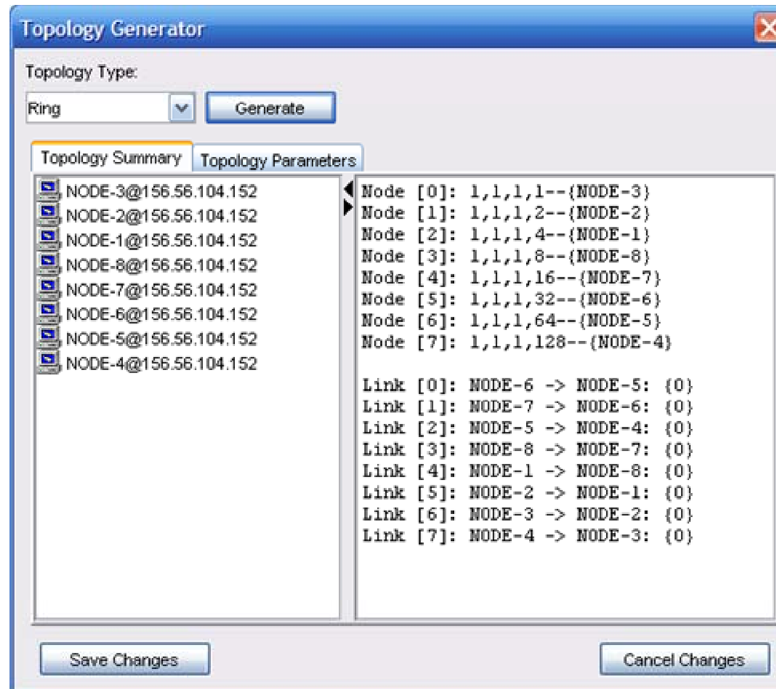


Figure 16: Nodes and Links Configuration for RING topology

4.7.7 Cluster topology

Cluster topology has more configuration parameters than the basic RING topology. These parameters [see Figure 17] define the characteristics of the generated topology such as the number of clusters, super-clusters and super-super-clusters.

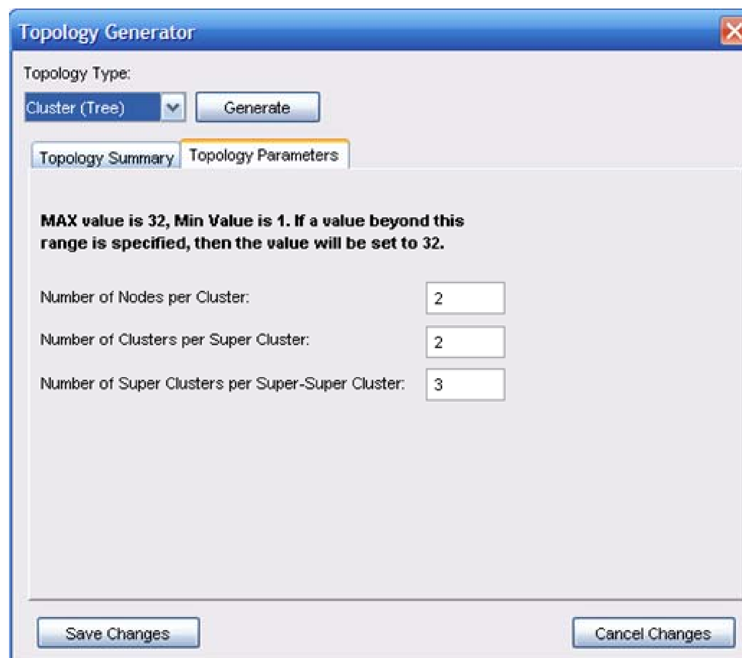


Figure 17: Cluster topology parameters

When **Generate** is clicked the output for an 8 node network using the above set parameters is shown in Figure 18. To complete the generation of the Topology and the linking-up of the nodes click **Save Changes,** and then **Commit** to generate the topology.

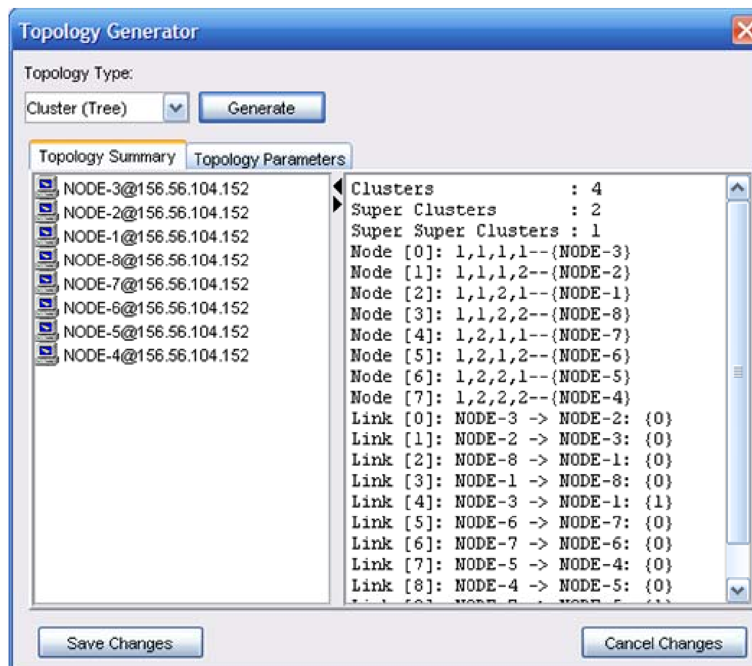


Figure 18: Nodes and Links Configuration for CLUSTER topology

4.7.8 Editing Links

The *Links* tab allows a user to edit pre-created links (via the topology generator) OR create / delete / modify user-defined links. Figure 19 shows the links created in an earlier run of CLUSTER topology generator. **NODE-3** has 3 out-going links.

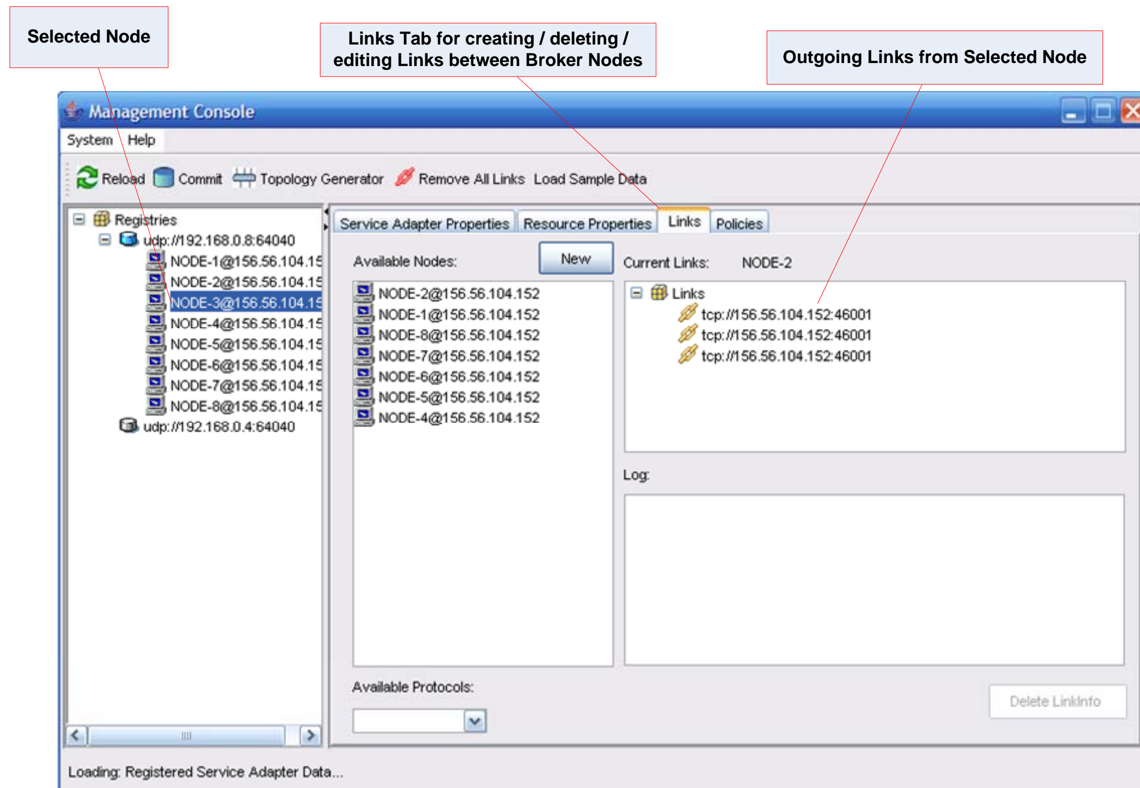


Figure 19: Editing Links

To delete an existing link, simply select the link to delete and click on **Delete LinkInfo** as shown in Figure 20. Be sure to **Commit** after you are done with the editing.

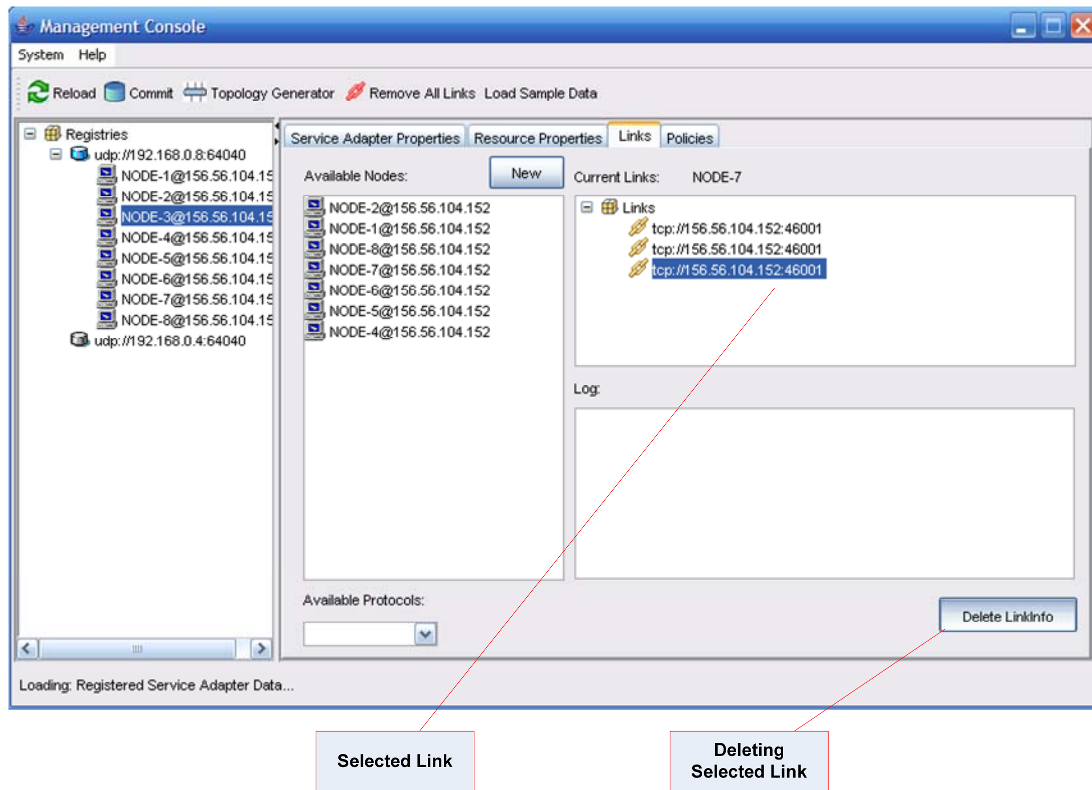


Figure 20: Deleting existing Links

4.7.9 Manually Creating Links

While creating links, the following must be noted

- Links can only be created between configured nodes. i.e. nodes which have been assigned properties in the Resource Properties tab after creating the node via Create Node.
- If the configuration changes after creating links, then the created links may not be deployed properly. This is because the link information contains physical IP addresses and port of the destination broker and this information is set when the link information is created. Thus, it is necessary to first set the broker configuration and then create the link.
- A link using a specific protocol between 2 nodes can only be created once and is directional, i.e. if a TCP link exists from NODE-1 to NODE-2, another TCP link from NODE-1 to NODE-2 cannot be created, however a TCP link from NODE-2 to NODE-1 can be created. Similarly an NIOTCP link between NODE-1 to NODE-2 can be created even if a TCP link was previously created.

The link creation process is illustrated in Figure 21. Once again, be sure to **Commit** the changes.

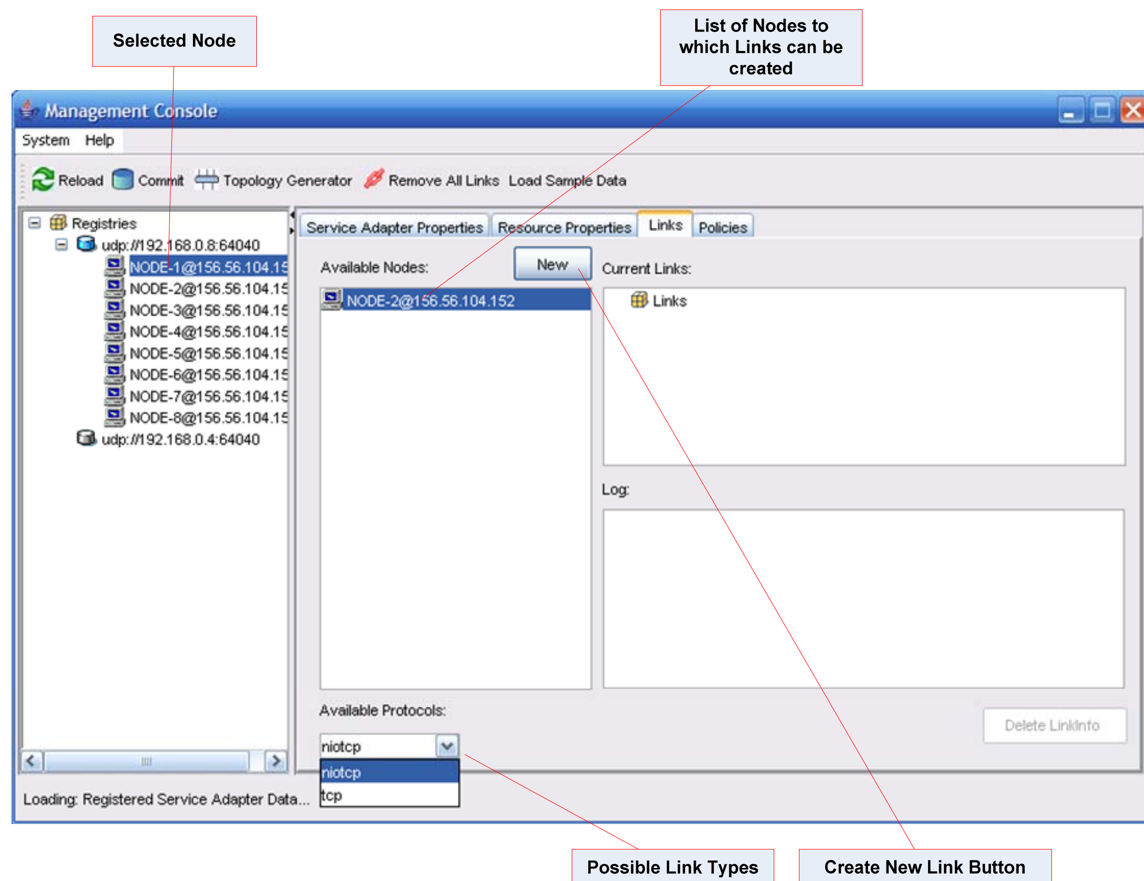


Figure 21: Manual Link Creation

To create a link from **NODE-1** to **NODE-2**, select **NODE-1** in the left pane. The Right pane's **Links** tab shows the available nodes. Available nodes are instantiated. Selecting an available node populates the available protocol list depending on the services configured on **NODE-2**. After selecting a protocol, simply click on the **New** button to create the link information for the link.

After nodes have been created and configured and the required link information set, the entire configuration information can be committed to registry by clicking the **Commit** button on the toolbar. The manager process associated with the nodes then picks up the configuration and deploys the network of brokers as defined by the user.

4.7.10Shutting down the Broker Network

To shutdown the broker network, simply go to the **Resource Properties** tab and click **Delete Node**. After the required nodes have been deleted, this information is committed to the registry by clicking the **Commit** button. The delete request is then acted upon by the respective manager processes.

5 Specifying the creation of Links

In this chapter we describe the creation of links in NaradaBrokering. We require that *properties* be specified for the creation of a link to any other NaradaBrokering node (broker or client alike). These properties snapshot information mandated by the NaradaBrokering transport layers to facilitate the creation of a communication link between 2 entities. This information generally pertains to –

a) The hostname and the port number which the process would be listening to
b) The underlying transport over which communications take place TCP, UDP, Multicast.
c) Whether data exchange should be over encrypted links.
d) Information required for tunneling through authenticating proxies and firewalls.

5.1 Creating a link

The properties specified for creation of links vary from transport to transport. In this section we describe the properties that need to be specified for the creation of different kinds of links to facilitate communications using different transport implementations. The properties always go in tandem with a specified transport type. The transport type is a String; examples of the transport type include "**`niotcp`**", "**`tcp`**", "**`udp`**", "**`multicast`**", "**`rtp`**", "**`ssl`**". We will also include a code snippet outlining the specification of these properties to create a link.

We now provide details pertaining to specifying properties for the creation of different types of links. We also outline the information encapsulated within these properties. For the purposes of our discussion lets assume that the connection is being initiated by a node A to another node B. Communication between two nodes over a certain transport type is predicated on the fact that both the nodes can support communications of the transport type in question.

Table 4: List of properties for specifying the creation of communication links in different transport protocols

Transport Type	Properties	Functions
TCP	TCPServerPort	Used for initialization of the `TCPLinkFactory`. When the value of this variable is set to 0 it implies that the node A initiating a connection to node B will not accept link creation requests from any other node.
	hostname	This is the hostname on which the node B's process is running.
	portnum	This is the port number on which node B accepts link creation requests from other nodes. In other words, the `TCPServerPort` specified to the

		TCPLinkFactory at node B is equal to portnum.
UDP	UDPListenerPort	The port on which node A would listen to communications.
	hostname	The host on which node B's process is running
	portnum	The port number on which node B is listening to datagram packets.
Multicast		No properties are required for setting up the MulticastLinkFactory.
	MulticastHost	To enable receipt & sending of data to a given multicast group.
	MulticastPort	To enable receipt &sending of data to a given multicast group.
RTP	RTPListenerPort	This needs to be specified for setting up the RTPLinkFactory. This is then used to exchange information pertaining to the RTP meeting id.
	dataPortLocal	To deal with raw RTP clients we need to establish two underlying communication paths. One is for the data packets and the other is for control packets. This is local port on which we listen for RTP data packets. A check is performed to see to it that this is an even number. Once this fact is confirmed we proceed to create another listener for the control packets at dataPortLocal+1.
	rtpHost	This is the host name on which the raw RTP Client resides.
	rtpPort	This is the port number on which the RTP client listens to for data packets. Once again this has to be even numbered. The raw RTP client listens to control packets on the rtpPort + 1. There are two underlying communication paths that are created by the specification of dataPortLocal, rtpHost and rtpPort. First, is the data path between dataPortLocal on node A to rtpHost and rtpPort on node B. The second is the control path between dataPortLocal+1 on node A to rtpHost and rtpPort+1 on node B.
SSL	truststore	Location of the trusted authorities database
	keystore	Location of the public/private key database
	truststorePassword	Password to the truststore

	keystorePassword	Password to the keystore
	username	The username for proxy authentication
	password	The password for proxy authentication
	domain	NT domain or workgroup for NTLM authentication
	host	local host name for NTLM authentication
	https.proxyHost	The location of the HTTPS proxy. Will try to auto detect from System properties if this does not exist.
	https.proxyPort	The location of the HTTPS proxy port. Will try to auto detect from System properties if this does not exist.
	secure	true | false. If false, will not do any real SSL.
	listenerport	The port to listen for incoming connections.
	host	The transport's end point's host name or IP address.
	port	The transport's end point's port number.

5.2 A Code Snippet Detailing Link Creation

The snippet below depicts the loading of properties to enable SSL, TCP and Multicast communications.

```
Properties props = new Properties();
 props.put("truststore", "D:/ SSLTunnel/keys/truststore");
 props.put("keystore", "D:/ SSLTunnel/keys/keystore");
 props.put("truststorePassword", "abc");
 props.put("keystorePassword", "abc");
 props.put("username", "test1");
 props.put("password", "test1");
 props.put("https.proxyHost", "everest");
 props.put("https.proxyPort", "8080");
 props.put("secure", "true");
 props.put("listenerport", "443");
 props.put("host", args[0]);
 props.put("port", args[1]);

 /** These properties pertain to setting up a blocking-TCP,
```

```
      and multicast link */
      props.put("hostname", args[0]); /** for both tcp, udp*/
      props.put("portnum", args[1]);  /** for both tcp, udp */
      props.put("TCPServerPort", "0"); /** for TCP */
      props.put("MulticastHost", "224.224.224.224");
      props.put("MulticastPort", args[1]);
```

5.3 Instructions for SSL/HTTPS connections through a proxy

To connect to the NaradaBrokering broker and client by using SSL/HTTPS over a proxy, please try the following steps:

(1) Add the keystore JVM parameter in the NaradaBrokering broker execution script available at **$NB_HOME/bin/startbr.sh**

```
java -Djavax.net.ssl.keyStore="/root/sslkeys/impromptu.localdomain.key" -Djavax.net.ssl.keyStorePassword=XX cgl.narada.node.BrokerNode $brokerConfigFile $serviceConfigFile $brokerCommunicatorPort&
```

(2) For coding the clients, the connection properties keystore and truststore are NOT used:

```
HTTPSconProp.put("trustStore", "c:/truststore");
HTTPSconProp.put("keyStore", "c:/keystore");
HTTPSconProp.put("trustStorePassword", "XXXXX");
HTTPSconProp.put("keyStorePassword", "XXXXX");
ini = new cgl.narada.jms.NBJmsInitializer(HTTPSconProp, "ssl");
```

(3) Instead the system properties should be set to point to a truststore

```
System.setProperty("javax.net.ssl.trustStore", DEFAULT_TRUSTSTORE);
System.setProperty("javax.net.ssl.trustStorePassword",DEFAULT_TRUSTPASS);
```

(4) In **$NB_HOME/config/ServiceConfiguration.txt**, there are two keystore parameters that are for the security framework, and do not have an effect on the SSL store requirements.

```
SecurityKeyStore=XXX
SecurityTrustStore=XXX
```

(5) In the **$NB_HOME/config/BrokerConfiguration.txt**, connect to the SSL broker port

```
SSLBrokerPort=443
```

5.4 Using IPSec

NaradaBrokering incorporates IPSec, allowing clients to traverse firewalls that prohibit other traffic. To enable this functionality, both the NaradaBrokering broker and clients must have additional software installed. Fortunately, this software is freely available and easily obtained.

We note that while IPSec is traditionally used to construct secure virtual private networks, we merely use IPSec as a tunnel to bypass firewalls for NaradaBrokering traffic. The IPSec connection established is not used for confidentiality or authenticity; upper layer protocols provide that security, when needed.

When implementing the IPSec connection documented below, the IPSec clients and servers will be deploying "split-tunneling." In this approach, a subset of the traffic from the machine will be tunneled through IPSec while the rest will be transmitted normally. In particular, the IPSec connection will only be used for traffic addressed to the other IPSec end-point. This allows NaradaBrokering to be used while not impacting other applications on the clients and servers.

IPSec can be used to traverse networks employing Network Address Translation (NAT). However, only the client can be behind NAT in the scenario documented below. If the server is behind a NAT, the Windows XP client machine must have a registry patch installed (see http://support.microsoft.com/default.aspx?kbid=885407). Unfortunately, IPSec is unlikely to work if multiple NATs are used between the client and server.

Below are instructions for configuring the client and server machines. The client instructions are written for Windows XP and the server is written for Fedora Linux. However, other Linux distributions can be used for the server while clients can additionally run MacOS X, Linux, and Windows 2000/Vista. Configuration support for these other versions will be added in subsequent releases.

5.4.1 IPSec Server:

To implement the IPSec server, we compile strongSwan 4.1 on Fedora 8 Linux. You can download strongSwan 4.1 from http://www.strongswan.org/download.htm . Before installing, you will need to ensure you have GCC and the GMP libraries (run "**`yum install gcc.i386 gmp.i386 gmp-devel.i386`**" as root).

To install strongSwan, uncompress the tarball, enter the extracted directory, run "**`./configure`**", "**`make`**", and "**`make install`**". If all the dependencies are met, this will install strongSwan system-wide.

The next step is to configure the strongSwan IPSec server. By default in Fedora, the configuration files are stored in **/usr/local/etc/**. Below is an **`ipsec.conf`** file that will

allow remote connections from all Windows IPSec machines using a *pre-shared secret*. This configuration file must be writable only by the root user. Please note: this configuration file is white-space sensitive! Lines must be indented with tabs as indicated and blank lines should only appear between the configuration setup and each connection definition.

Table 5: The ipsec.conf configuration file

```
--- BEGIN Configuration File - ipsec.conf --- config setup
	nat_traversal=yes
	charonstart=no

conn CGL-IPSec
	authby=secret
	pfs=no
	rekey=no
	keyingtries=3
	# -------------------------------------------------------
	# The VPN server.
	#
	# Allow incoming connections on the external network interface.
	# If you want to use a different interface or if there is no
	# defaultroute, you can use:   left=your.ip.addr.ess
	#
	left=%defaultroute
	#
	# Required for Windows XP:
	leftprotoport=0/%any
	#
	# -------------------------------------------------------
	# The remote user(s).
	#
	# Allow incoming connections only from this IP address.
	#    Use right=%any to allow any incoming connections.
	right=%any
	#
	# Same thing as the leftprotoport, only for the remote user:
	rightprotoport=0/%any
	#
	# -------------------------------------------------------
	# Actually enable this configuration:
	auto=add
--- END Configuration File - ipsec.conf ---
```

In addition to the basic configuration file, the server must have a list of pre-shared secrets to authenticate the remote client. These are stored in the **ipsec.secrets** file (again located in **/usr/local/etc/** by default in Fedora). This file must be read and writable only by the root user.

Table 6: The ipsec.secrets configuration file

```
--- BEGIN Configuration File - ipsec.secrets --- w.x.y.z %any: PSK
"shared_secret_goes_here"
--- END Configuration File - ipsec.secrets ---
```

Note that "**w.x.y.z**" is replaced with the IPv4 or IPv6 IP address of the server. The **"%any**" specifies any client can connect; it can be replaced with a specific address to restrict the acceptable clients. The value in quotation marks is replaced with the shared secret that clients must supply to connect to the server.

Once the configuration phase is completed, simply run "**ipsec start**" as root, which will allow clients to begin connecting using IPSec. To see established connections, you can run "**ipsec status**" for a concise output or "**ipsec statusall**" for detailed output.

By default, strongSwan will write its log file to **/var/log/secure** in Fedora. This is useful for troubleshooting and to monitor connections. Adding "**plutodebug=control**" to the "**config setup**" section of the **ipsec.conf** file will increase the verbosity of the connection process logging. For further IPSec troubleshooting, see http://www.strongswan.org/support.htm . Additionally, the OpenSwan project, which forked from the same base code as strongSwan, has documentation available at http://wiki.openswan.org/index.php/ , which may provide some support.

5.4.2 IPSec Client:

In the current documentation for IPSec capabilities within NaradaBrokering, we focus on Windows XP. However, Windows 2000 (Service Pack 3 or higher) and Vista also provide IPSec and will be documented in future releases. For Windows XP, either Service Pack 1 or 2 must be installed.

For users of Service Pack 1, a patch must be downloaded from Microsoft for Network Address Translation Traversal (NAT-T), which is required if the IPSec client is behind a NAT. This patch is available at http://support.microsoft.com/support/kb/articles/q818/0/43.asp . Windows XP Service Pack 2 users already have this patch installed.

To use IPSec, you must install the ipseccmd.exe tool. This is available on the Windows XP CD under Windows Support Tools. Windows XP Service Pack 2 users should download an updated version of the support tools from Microsoft at http://support.microsoft.com/default.aspx?scid=kb;en-us;838079 .

Once this is installed, IPSec will be available from the command line in Windows XP. NaradaBrokering will invoke this for you as needed. Once the IPSec tunnel policy has been specified, a connection must be established using the machine. The actual IPSec tunnel is established on demand. Accordingly, the first packet transmitted to the destination will begin the IPSec tunnel creation. Typically, the ping command is used to establish the tunnel. Below is output that you may see when pinging the client after executing the IPSec command:

Table 7: Output of the ping command after executing the ipsec command

```
Pinging w.x.y.z with 32 bytes of data:

Negotiating IP Security.
Negotiating IP Security.
Reply from w.x.y.z: bytes=32 time<1ms TTL=64
Reply from w.x.y.z: bytes=32 time<1ms TTL=64

Ping statistics for w.x.y.z:
    Packets: Sent = 4, Received = 2, Lost = 2 (50% loss)
```

From this we see that the first two ping packets were lost while the IPSec tunnel was being created, which is a one-time packet loss. However, the last two packets were transmitted under the IPSec tunnel. Subsequent packets will be transmitted under IPSec until the tunnel is deactivated.

To see what IPSec policies have been configured, you can run "**ipseccmd show all**" on the client machine.

When you have finished with the IPSec tunnel, you can deactivate it using the following command:

--- BEGIN IPSec Tunnel Termination Command --- **ipseccmd -w REG -p "NaradaBrokering Policy" -r "NaradaBrokering Rule" -y**
--- END IPSec Tunnel Termination Command ---

The tunnel will then be deactivated and subsequent traffic will be sent without IPSec.

For more information on the ipseccmd.exe command, please see http://www.microsoft.com/resources/documentation/windows/xp/all/proddocs/en-us/ipsecmd.mspx

5.4.3 Setting up of the IPSec Tunnel from NaradaBrokering clients

NaradaBrokering allows you to set up IPSec tunnels (and subsequently tear them down) from your program. To do so, during the creation of link one needs to specify a set of properties, some of which are mandatory and some of which have default values assigned by NaradaBrokering. These properties are listed in the table below.

Table 8: Properties to be specified for creation of IPSec Tunnel

Property Name	Comments
IPSecHostname	[mandatory] Creates a filter for all traffic between the current machine and the IPSec server ("w.x.y.z"). Only the traffic between these two machines will enter the IPSec tunnel.
IPSecSharedSecret	[mandatory] Authenticates the server using a pre-shared secret of *shared_secret_goes_here*. This **must match** what is stored in the server's ipsec.secrets file or the client will be unable to connect.
IPSecPolicyName	[default: NaradaBrokering Policy] Names the policy that is being created. This is later needed to deactivate the tunnel.
IPSecRuleName	[default: NaradaBrokering Rule] Names the rule that is being added to the policy. This is needed later on to deactivate the tunnel.

The cryptographic primitives used to secure the IPSec tunnel are 3DES and MD5 since both Windows and the strongSwan server supports them. While both these primitives are considered weak, we are using the IPSec tunnel only for firewall traversal and not for security purposes.

The code snippet below describes the creation of IPSec links. The NaradaBrokering transport layer handles all the complexity of setting up the tunnel, issuing the ping command to

ensure packets are not lost subsequently, and finally the tearing down of the tunnel once the link to the broker is closed.

```
props.put("IPSecHostname", args[0]);
props.put("IPSecSharedSecret", "shared_secret_goes_here");

clientService.initializeBrokerCommunications(props, "ipsec");
```

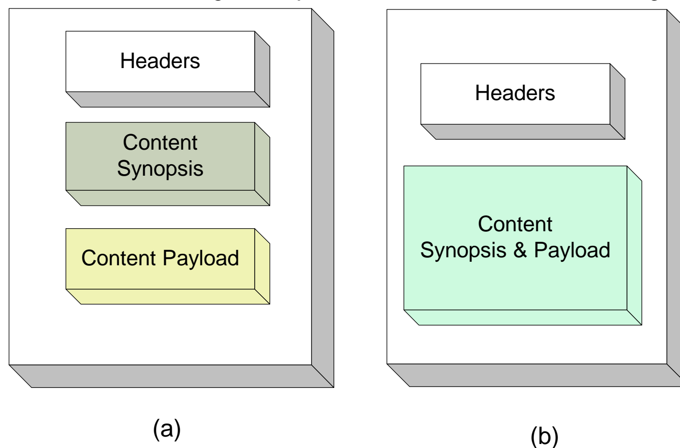
Note that once, the IPSec Tunnel has been established all registered transports within NaradaBrokering can utilize this tunnel for communications with the broker over different protocols. Once the client closes its connection to the broker, the IPSec tunnels are automatically torn down by NaradaBrokering.

6 Developing NaradaBrokering Applications

In this chapter we introduce some basic concepts that are important to the development of applications in NaradaBrokering. We then proceed to develop a simple application in NaradaBrokering.

6.1 Primer on events, synopsis, profiles and templates

An event comprises of headers, content descriptors and the payload encapsulating the content. An event's headers provide information pertaining to the type, unique identification, timestamps, dissemination traces and other QoS related information pertaining to the event. The content descriptors for an event describe information pertaining to the encapsulated content. The content descriptors and the values that these descriptors take, collectively comprise the event's *content synopsis*.



Depending on how elaborate and complex the content description process is it is sometimes conceivable that the demarcation between synopsis and the content is blurred and that they end up being indistinguishable from each other. For example an XML event's synopsis may conceivably describe all the content, while the content may be dispersed across these content descriptors.

Entities have multiple *profiles* each of which signifies an interest in events conforming to a certain template. This interest is typically specified in the form of a constraint that events need to satisfy, before being considered for routing to the entity in question. This constraint is also sometimes referred to as a subscription. Entities specify constraints on the content descriptors and the values some or all of these descriptors might take. Individual profiles can also include information pertaining to the device type – CPU capability, and security related information that would sometimes be needed for special processing of events.

When an event traverses through the system these constraints are evaluated against the event's synopsis to determine the eventual recipients. An event's synopsis thus determines the entities that an event needs to be routed to. Two synopses are said to be equal if the content descriptors and the values these descriptors take are identical. It is possible for events with the same synopsis to encapsulate different content in its payload. It is however possible for events with different synopses to be routed to the same set of destinations. An event's synopsis also contains information pertaining to the originator of the event.

The type of constraints specified by the entities varies depending on the complexity and type of the content descriptors. Examples of content descriptors include a simple character string describing event topic information, an XML document with various elements and nodes describing content elaborately, and finally a set of properties that can be set to different values depending on the content. In each of these aforementioned cases the constraints specified would be different – a simple character string based equality test, an XPath query on the XML document and an SQL like query on the properties and the values these properties take.

6.2 Writing a simple NaradaBrokering client

6.2.1 Initializing the Client Service

The developer first needs to specify an identifier for the client. Currently this is an integer value. We are proposing to replace this by UUIDs. Next, one needs to control the configuration of the clients. See the **$NB_HOME/config/ServiceConfiguration.txt** for a sample configuration file. This file is used to set up and control parameters needed by various services. Default values are used if the correct file is not specified.

Table 9: Initializing the ClientService

```
int entityId = 7878;

String config = "/NaradaBrokering-x.y.z/config/ServiceConfiguration.txt";
SessionService.setServiceConfigurationLocation(config);

ClientService clientService = SessionService.getClientService(entityId);
```

The code snippet below, demonstrates two functions. First, one can initialize the configurations associated with the various services in one's session. Second, the `ClientService` instance is initialized based on the specified `entityId`. The method calls listed in the code snippet below will throw an exception- `cgl.narada.service.ServiceException` – if it encounters problems.

6.2.2 Initializing communications with the broker

Once the `ClientService` has been initialized, we need to set up communications with the broker. NaradaBrokering incorporates supports for several different transport protocols (as described in section 5) and a given broker typically listens to communications over several different ports (section 2). Setting up communications with the broker involves specifying information about the host, port and other elements such as configuration and security related information.

It is very simple to initialize and load communication libraries in NaradaBrokering clients. One first needs to create a `java.util.Properties` object and load the appropriate values for the various elements that are needed by a given protocol. Table 4 provides a list of the properties that are expected (and can be specified) for different transport protocols.

Table 10: Initializing communications with the broker

```
Properties props = new Properties();
props.put("hostname", "localhost");
props.put("portnum", "3045");

clientService.initializeBrokerCommunications(props, "niotcp");
```

The code snippet (Table 10) demonstrates the initialization of communications with the broker using non-blocking TCP; the snippet outlines initializations for the case where the broker is running on **`localhost`** and listening to socket connections on port **`3045`**. The transport type `String` which is specified as the second argument of the **`initializeBrokerCommunications()`** method is case-sensitive. Other examples of valid transport types include "**`niotcp`**", "**`tcp`**", "**`udp`**", "**`multicast`**", "**`rtp`**", "**`ssl`**" and ""**`ipsec`**".

6.2.3 Initializing the consumer role

A consumer is an entity that is interested in consuming messages. Every consumer needs to implement the **`cgl.narada.service.client.NBEventListener`** interface. This interface contains the **`onEvent(NBEvent nbEvent)`** method that is invoked by the substrate upon receipt of a message which should be routed to that consumer. To create a consumer and register it with the NaradaBrokering substrate one needs to use the following code snippet. Note that the **`this`** in the code-snippet above refers to the class, which implements the `NBEventListener` interface.

Table 11: Initializing the consumer

```
EventConsumer consumer = clientService.createEventConsumer(this);
```

The next step involves specifying the subscription, which clearly specifies the type of messages the consumer is interested in consuming. The NaradaBrokering substrate incorporates support for a wide-variety of subscription formats, these include: "/" separated String, "," separated <tag, value> pairs, Regular expressions, XPath etc.

In our currently example we deal with the simplest form which is String based. To subscribe, we first need to create the Profile. We then use the EventConsumer that we created to subscribe to the profile. The code snippet outlining the steps we described here is depicted in Table 12.

Table 12: Initializing subscriptions

```
int profileType = TemplateProfileAndSynopsisTypes.STRING;
Profile profile =
   clientService.createProfile(profileType, "Movie/Casablanca");

consumer.subscribeTo(profile);
```

The system places not limits on the number of consumers that can be created from a given ClientService instance, nor is there any limit on the number of subscriptions that you can subscribe to on a given EventConsumer.

Events that an entity receives are delivered using the **onEvent(NBEvent nbEvent)** method. The processing logic associated with received events can be put in this method. An entity can inspect the received **NBEvent** event to retrieve its headers, synopsis, payloads etc. In the simplest case, you can print the event's payload. The code below depicts the simplest implementation of the **onEvent()** method.

Table 13: A simple implementation of the onEvent() method

```
public void onEvent(NBEvent nbEvent) {
    String synopsis = (String) nbEvent.getContentSynopsis();
    System.out.println(moduleName + "Received NBEvent {" + synopsis + "} "
       + new String(nbEvent.getContentPayload()));
}
```

6.2.4 Initializing the producer role

A producer is responsible for the generation of streams. If an application needs such a producer of events, an `EventProducer` needs to be initialized. This is done by simply invoking the method **`createEventProducer()`** on the `ClientService`.

Additionally, if an application operates in a dual role as a subscriber to events of a stream to which it producer role publishes to, it can configure the aforementioned producer to suppress redistribution of events back to the application. The `EventProducer` interface also allows one to configure the generation of identifiers, inclusion of template identifiers, disabling the generation of timestamps etc.

The code snippet below (Table 14) depicts the initialization of the EventProducer and configuration of some of the capabilities.

Table 14: Initializing the producer and configuring capabilities

```
EventProducer producer = clientService.createEventProducer();

producer.setSuppressRedistributionToSource(true);

producer.generateEventIdentifier(true);
producer.setTemplateId(12345);
producer.setDisableTimestamp(false);
```

Once the producer has been initialized we can proceed to the generation of events. To generate events, one needs to specify the *event type*, the *content synopsis* and the *payload* for the event. Once the event has been created, the producer can publish the created event. This is depicted in the code snippet below (Table 15).

Table 15: Publishing messages using the `EventProducer`

```
int eventType = TemplateProfileAndSynopsisTypes.STRING;
String synopsis = "Movie/Casablanca";
byte[] payload;

NBEvent nbEvent = producer.generateEvent(eventType, synopsis, payload);

producer.publishEvent(nbEvent);
```

6.2.5 Event Properties

In addition to headers and payloads, messages in NaradaBrokering can have `EventProperties` associated with them. These properties are user-defined and come in two flavors: mutable and immutable.

Immutable properties cannot be changed once they have been set. Mutable properties can be modified several times. Furthermore, not only can one can track the property changes that have occurred on a mutable property, they can also track the entities that initiated these changes.

The code snippet below (Table 16) outlines the generation and processing of events with `EventProperties`.

Table 16: Working with `EventProperties`

```
public void addEventProperties(NBEvent nbEvent) {
    EventProperties properties = nbEvent.getEventProperties();
    properties.setMutableProperty("Tennis", "US Open",
                                  new Integer(entityId));
}


public void processEventProperties(NBEvent nbEvent) throws NBEventException {
    if (nbEvent.hasEventProperties()) {
       EventProperties eventProperties = nbEvent.getEventProperties();
       Enumeration propertyNames = eventProperties.getPropertyNames();

       while (propertyNames.hasMoreElements()) {
          Object _propertyName = propertyNames.nextElement();
          String propertyType =
          eventProperties.isMutable(_propertyName) ? "mutable": "immutable";
          System.out.println("Property [" + _propertyName
             + "] was last modified by "
             + eventProperties.getLastModifier(_propertyName)
             + ". This is a " + propertyType + " property");
            }
        }
    }
```

6.3 Harnessing the available Qualities of Services

NaradaBrokering incorporates support for several Quality of Services (QoS). In this section we will disccus how applications can harness some of these QoS. Specifically, we will address how an application can avail of services related to compression, fragmentation, reliable delivery, and replay that are available within NaradaBrokering. Generally, harnessing the QoS involves the creation of `ProducerConstraints` and `ConsumerConstraints`, which are associated with the publishing and consumption of events respectively.

6.3.1 Consumer Constraints

`ConsumerConstraints` allow a consumer to specify QoS constraints on the receipt of events conforming to a given profile. `ConsumerConstraints` are created by the `EventConsumer` by using the `Profile` on which the constraints are to be specified; this QoS constraint on the subscription is then propagated. The `ConsumerConstraints` interface contains several methods that allow one to configure various aspects of the QoS being harnessed. The code snippet below (Table 17) depicts the creation of the constraints and the process of registering these constraints to a specific subscription profile.

Table 17: Registering `ConsumerConstraints` to a specific `Profile`

```
EventConsumer consumer;
// Initialization of the EventConsumer has been elided for clarity

ConsumerConstraints constraints= consumer.createConsumerConstraints(profile);

consumer.subscribeTo(profile, constraints);
```

6.3.2 Producer Constraints

`ProducerConstraints` allow a producer to specify QoS constraints on the generation of events that conform to a specific template. `ProducerConstraints` first require the creation of a `TemplateInfo`; this requires the specification of the `templateId`, `templateType` and `template`. Once the `TemplateInfo` has been created, an instance of `ProducerConstraints` can then be created by the `EventProducer`. The `ProducerConstraints` interface contains several methods that allow one to configure various aspects of the QoS being harnessed. The code snippet below (Table 18) depicts the process of creating the `TemplateInfo` and from thereon the `ProducerConstraints` and

its use in specifying constraints associated with individual events: the constraints can be specified on a per-event basis.

Table 18: Generation of `ProducerConstraints` and publishing events

```
int templateId = 12345;
int templateType = TemplateProfileAndSynopsisTypes.STRING;
Object template = "Movie/Casablanca";
TemplateInfo templateInfo =
     clientService.createTemplateInfo(templateId, templateType, template);


EventProducer producer;
// Initialization of the EventConsumer has been elided for clarity

ProducerConstraints producerConstraints =
     producer.createProducerConstraints(templateInfo);

producer.publishEvent(nbEvent, producerConstraints);
```

6.3.3 Compression and Decompression Services

In this section we describe how applications can utilize the compression and decompression services available within NaradaBrokering. These services are among the simplest QoS available for applications within the substrate.

In this case the QoS constraints are associated only with the producer. The producer creates the `ProducerConstraints` and also the `Properties` encompassing the algorithm to be used for the compression of payloads. Upon encountering a compressed payload, the system automatically decompresses the payloads prior to delivery to the relevant consumers. The code snippet below (Table 19) demonstrates how the producer initializes compression capabilities.

Table 19: Utilizing the compression and decompression services

```
Properties compressionProperties =  new Properties();
compressionProperties.put("compressionAlgo", "zlib");

producerConstraints.setSendAfterPayloadCompression(compressionProperties);

producer.publishEvent(nbEvent, producerConstraints);
```

6.3.4 Reliable Delivery Services

To utilize this service, the user first needs to set up a Repository Node separately. This process is described in detail in section 7. The Repository Node has been tested with MySQL and PostgreSQL. The remainder of the discussions in the section proceeds under the assumption that this node has been set up based on the specified instructions.

6.3.4.1 Initializing the consumer

We start by first focusing on the consumer that is interested in harnessing the reliable delivery service. To do so, we first need to create the appropriate `ConsumerConstraints`, and then invoke appropriate methods on these constraints to configure reliable delivery properties. Finally, we associate these constraints with the appropriate subscription profile. The code snippet (Table 20) outlines the steps involved in initializing an `EventConsumer` that is interested in consuming messages reliably.

Table 20: Initializing constraints for consuming reliably

```
ConsumerConstraints constraints =
     consumer.createConsumerConstraints(profile);

constraints.setReceiveReliably(templateId);

consumer.subscribeTo(profile, constraints);
```

A key feature of the reliable delivery service is to be able to retrieve events after a *failure* or a *disconnect*. To avail of this feature the application needs to implement the **`cgl.narada.service.client.NBRecoveryListener`** interface, and initiate recovery by invoking the **`recover()`** method on the `EventConsumer` that has registered for reliable delivery.

Table 21: Initiating recovery of the consumer

```
public class RobustApp implements NBEventListener, NBRecoveryListener {

   long recoveryId= consumer.recover(templateId, this);

   //Upon completion of the attempt to recover, this method is invoked on the
   //listener that was registered with the */
   public void onRecovery(NBRecoveryNotification recoveryNotification) {
     System.out.println(recoveryNotification);
   }

}
```

The operations involved in initiating the recovery of the consumer are depicted in the code snippet (Table 21); the **this** in the code snippet corresponds to the Java Class that implements the aforementioned NBRecoveryListener interface, which is used to notify a consumer of the status of the recovery process that it initiated.

6.3.4.2 Initializing the producer

In our next step, we focus on ensuring that the EventProducer performs certain actions that ensure that messages are generated reliably. To do so, we first need to create the appropriate ProducerConstraints, and then invoke appropriate methods on these constraints to configure the reliable delivery properties. Finally, we associate these constraints with the events that we need to publish reliably. The code snippet (Table 22) outlines the steps involved in initializing an EventProducer that is interested in producing messages reliably.

Table 22: Initializing ProducerConstraints for producing reliably

```
TemplateInfo templateInfo =
     clientService.createTemplateInfo(templateId, templateType, template);

 producerConstraints = producer.createProducerConstraints(templateInfo);
 producerConstraints.setSendReliably();

 producer.publishEvent(nbEvent, producerConstraints);
```

To reinitialize the producer after a failure or disconnect one needs to implement the **NBRecoveryListener** interface, and initiate recovery by invoking the **recover()** method. This is depicted in the code snippet (Table 23); the **this** in the code snippet corresponds to the Java Class that implements the aforementioned NBRecoveryListener interface, which is used to notify a producer of the status of the recovery process that it initiated..

Table 23: Initiating recovery of the EventProducer

```
public class RobustApp implements NBEventListener, NBRecoveryListener {

   long recoveryId= producer.recover(templateId, this);

   //Upon completion of the attempt to recover, this method is invoked on the
   //listener that was registered with the */
   public void onRecovery(NBRecoveryNotification recoveryNotification) {
     System.out.println(recoveryNotification);
   }

}
```

6.3.5 Managing Replays

The Replay Service works with events that have been stored reliably using the Reliable Delivery Service the use of which was described in the preceding section. The Replay Service is used by the consumers to play back events that were *previously* archived. There are three steps involved in utilizing the replay service. The first, involves generating events reliably. The second step involves creating the appropriate `ReplayRequest` object which encapsulates the set of events that need to be played back. The final step, involves initiating the replay based on the created `ReplayRequest`.

6.3.5.1 Creating the appropriate ReplayRequest

In the section, we focus on the creation of a `ReplayRequest`, which encapsulates a request that initiates playbacks. A replay request could be based on the following:

- A specified set of sequence numbers
- A specified range of sequence numbers
- A specified range of sequence numbers that also includes additional constraints that need to be satisfied by the events that will be played back.

The `ClientService` interface provides these methods to create the `ReplayRequest`.The code snippet (Table 24) outlines two different ways to create these requests both of which require the **`templateId`** to be specified. The first request is created based on sequence numbers, while the second one is based on a range (specified by the **`start`** and **`end`** values) that has been specified.

Table 24: Creating a ReplayRequest

```
long[] sequenceNumbers;
// Initialize the sequences to be played …

ReplayRequest replayRequest =
     clientService.createReplayRequest(templateId, sequenceNumbers);

long start, end;
//Initialization of start and end

ReplayRequest replayRequest2 =
    clientService.createReplayRequest(templateId,  start, end);
```

In the third approach, one specifies the **`templateId`**, the range of sequences to be replayed, along with any additional profile constraints for delivery.

6.3.5.2 Initiating Replays

The `ReplayRequests` which were created in the previous section are used to initiate replays. In order to be able to initiate replays, the application first needs to implement the **`cgl.narada.service.replay.ReplayServiceListener`** interface. This interface has two methods which are used by the substrate to playback events and also to report on the status of previously issued `ReplayRequests` respectively:

- `public void` **`onReplay`**`(ReplayEvent replayEvent)`
- `public void` **`onReplayResponse`**`(ReplayResponse replayResponse)`

The code snippet (Table 25) outlines the process of initiating replays; the **`this`** in the code snippet corresponds to the Java Class that implements the `ReplayServiceListener` interface. The methods related to processing the `ReplayEvents` and the responses to the `ReplayRequest` have been elided for clarity.

Table 25: Initiating a Replay

```
public class ReplayApp implements  ReplayServiceListener {

   public void initializeConsumer() throws ServiceException {
     consumer = clientService.createEventConsumer(this);
   }

   public void performReplay(int templateId, long start, long end)
     throws ServiceException {
     ReplayRequest replayRequest =
       clientService.createReplayRequest(templateId, start , end );

     consumer.initiateReplay(replayRequest, this);
  }


  /** Process the playback event */
   public void onReplay(ReplayEvent replayEvent) {
       ....
   }

   /** Process the response to a previously issued ReplayRequest */
   public void onReplayResponse(ReplayResponse replayResponse) {
     ....
   }
}
```

6.3.6 Fragmentation and Coalescing

In this section we describe services for fragmenting large payloads, and the inverse service, the coalescing service, for reconstituting these fragments into the original payload. When these services work in tandem we are able to break up large payloads (typically files) into smaller fragments and reliably coalesce them at the consumer.

This scheme was used in the NaradaBrokering-enhanced version of GridFTP. This scheme allowed us to initiate file transfers without the recipient even being present at the time the file transfer was taking place. Furthermore, this also allows one-to-many transfers. The fragmentation/coalescing services require the NaradaBrokering Reliable Delivery Service which was discussed in previous sections. Please see the **%NB_HOME%/config/ServiceConfiguration.txt** configuration file to configure the parameters related to these. This includes the location of the temporary directories that are needed by these services to store the fragments.

6.3.6.1 The Fragmentation Service

In this section we focus on the fragmentation service. Specifically, we are interested in ensuring that the EventProducer performs certain actions that ensure that it is able to fragment the payload correctly. To do so, we first need to create the appropriate ProducerConstraints, and then invoke appropriate methods on these constraints to configure the fragmentations properties. The fragmentation properties take two sets of parameters. One can specify either one of these sets as the fragmentation properties.

- **fileLocation** and **fragmentSize**. This controls the size of the fragments for the specified file.
- **fileLocation** and **numOfFragments**. This controls the total number of fragments for a given file.

Table 26: Initializing the EventProducer to fragment large payloads

```
public class FragmentApp implements NBRecoveryListener {

   long recoveryId= producer.recover(templateId, this);

   public void initializeProducerConstraints() {
      Properties fragmentationProperties = new Properties();
      fragmentationProperties.put("numberOfFragments", 300);
      fragmentationProperties.put("fileLocation", filename);
      producerConstraints.setSendAfterFragmentation(fragmentationProperties);
   }

   public void initiateFragmentation() {
     producer.publishEvent(nbEvent, producerConstraints);
   }
```

```
   //Upon completion of the attempt to recover, this method is invoked on the
   //listener that was registered with the */
   public void onRecovery(NBRecoveryNotification recoveryNotification) {
     System.out.println(recoveryNotification);
   }

}
```

Finally, we associate these constraints with the events whose payload we need to fragment. The code snippet (Table 26) outlines the steps involved in initializing an `EventProducer` that is interested in fragmenting a large payload and subsequently using these constraints to initiate the fragmentation of the file and subsequent transfer to consumers that are interested in the receipt of this file. The next section will describe how one can avail of the coalescing service to reconstitute these fragments.

6.3.6.2 The Coalescing Service

In this section we focus on the consumer that is interested in coalescing fragments produced by the fragmentation service. To do so, we first need to create the appropriate `ConsumerConstraints`, and then invoke appropriate methods on these constraints to configure reliable delivery properties. Finally, we associate these constraints with the appropriate subscription profile. The code snippet (Table 27) outlines the steps involved in initializing an `EventConsumer` that is interested in coalescing fragments.

Table 27: Initializing the `EventConsumer` to coalesce fragments of a large payload

```
public class CoalescingApp implements NBEventListener, NBRecoveryListener {

   long recoveryId= producer.recover(templateId, this);

    ConsumerConstraints constraints
          =consumer.createConsumerConstraints(profile);
    constraints.setReceiveReliably(templateId);
    constraints.setReceiveAfterCoalescingFragments();

    consumer.subscribeTo(profile, constraints);

    long recoveryId = consumer.recover(templateId,this);


   //Upon completion of the attempt to recover, this method is invoked on the
   //listener that was registered with the */
```

```
   public void onRecovery(NBRecoveryNotification recoveryNotification) {
     System.out.println(recoveryNotification);
   }

}
```

Note that the large file will be coalesced in the directory specified in the **%NB_HOME%/config/ServiceConfiguration.txt** configuration file. The large coalesced file will not be maintained in main memory; instead, the consumer will receive a notification indicating that the precise location of the file.

7 Setting up the Repository Node

The first thing to make sure is that the MySQL database has been installed. MySQL can be downloaded from http://www.mysql.com, make sure that you download a stable version as recommended by the website and install it. NaradaBrokering's RobustNode has been tested with version's 3.23 through 4.1 of the MySQL database.

7.1 Creating the Database and Tables (Windows and Linux)

If you are using the Windows operating system, the database and tables can easily be created by using the **bat** files that have been provided in the distribution. These bat files are located in the "**NB_Home/bin/mysqlCommands**" directory. If your database access requires the specification of a user name and password, you need to modify "bat" files as instructed within these **bat** files to make sure that you specify the user name and password while running the **mysql** command viz. **mysql -u username -p**. The mysql program will then ask you for your password.

To create database and its tables, execute following bat files sequentially.
CreateDatabase.bat
CreateTables.bat

If you want to delete database and/or tables, you can use following files:
DropTables.bat
DropDatabase.bat

Note that the **sql** files that have provided are platform independent, and will work with both Windows and Unix environments.

For the Linux operating system, use the aforementioned SQL files to create and drop the database and/or its tables:

create_database.sql
create_tables.sql
drop_database_sql
drop_tables.sql

To simplify management of your database, you can download the MySQL Control Center from its site. It provides a GUI interface to manage users, databases and tables. We suggest creating a new user account to access the database instead of using the "root" account.

Finally, using a *compatible* JDBC driver is very important. Your driver must support your MySQL database to create JDBC connection. If you get an exception, please check your driver compatibility.

7.2 Using the Robust Node

Step 1: Double click **startBroker.bat** . This starts the broker.

Step 2: Next, double click **robustNode.bat** . This starts robust delivery service and the underlying stable storage. The GUI for this application is depicted below.

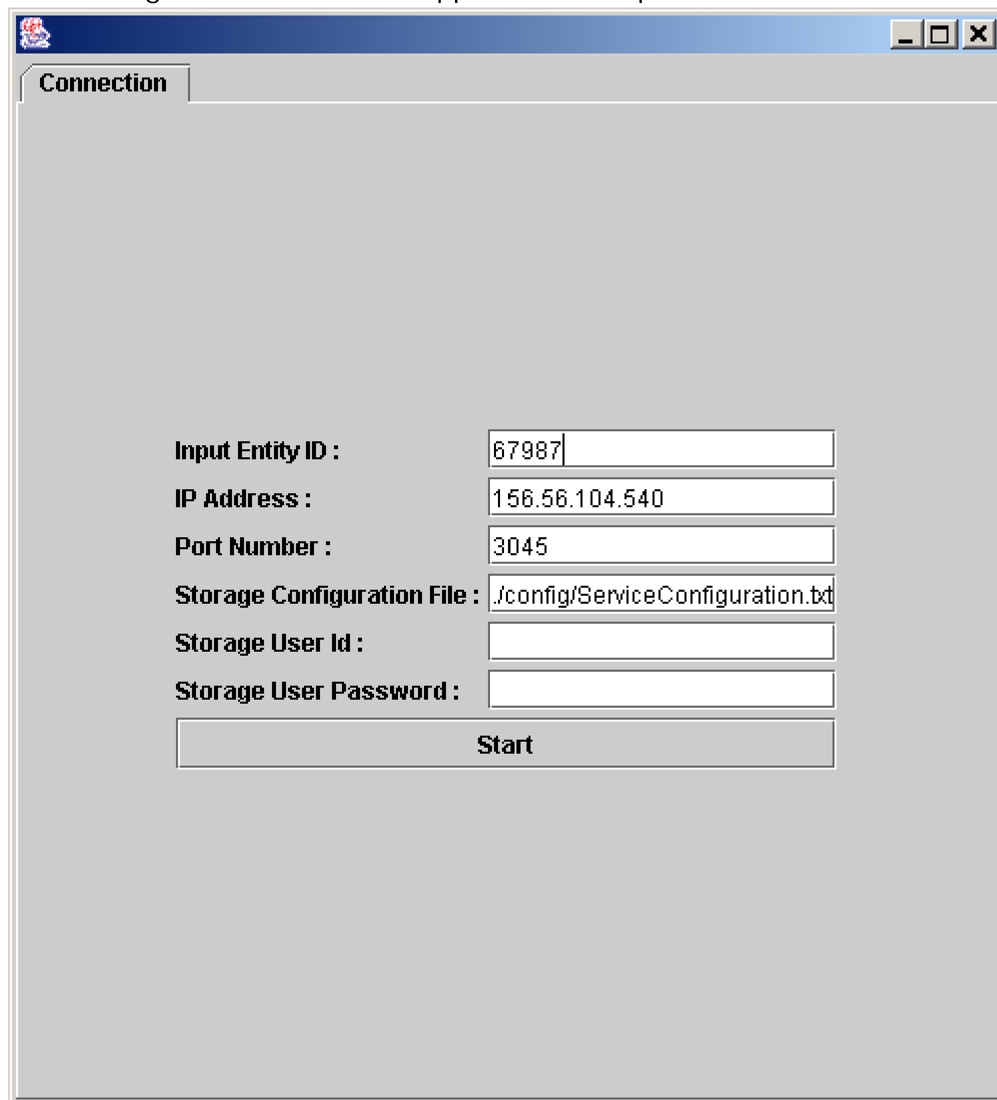


Figure 22: The opening screen

Specify the broker-**IP address** and the **Entity Id** (it could be anything e.g. 888) for the robust node. Proceed to press the **Start** button. If all goes well in the connection to the broker and the setup of the JDBC connections you will see the GUI depicted in Figure 23.

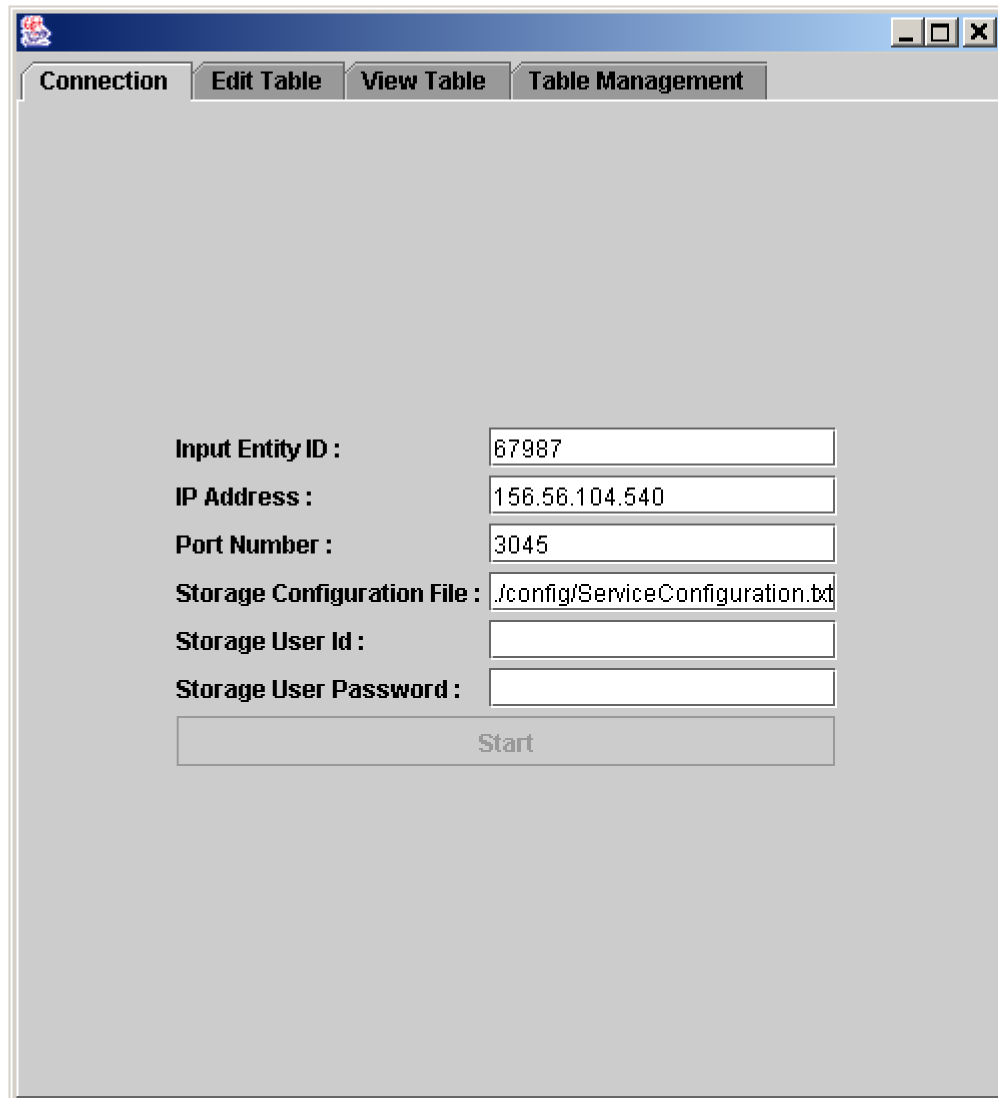


Figure 23: The GUI after the successful setup of the broker connection and the storage service

Proceed to click the **View Table** tab. If this is the first time that you are running this application everything will be empty and there will be no entries in the sub tabs that are available e.g. **Inventory**, **Template**, **Profile** and **EntityTemplate**.

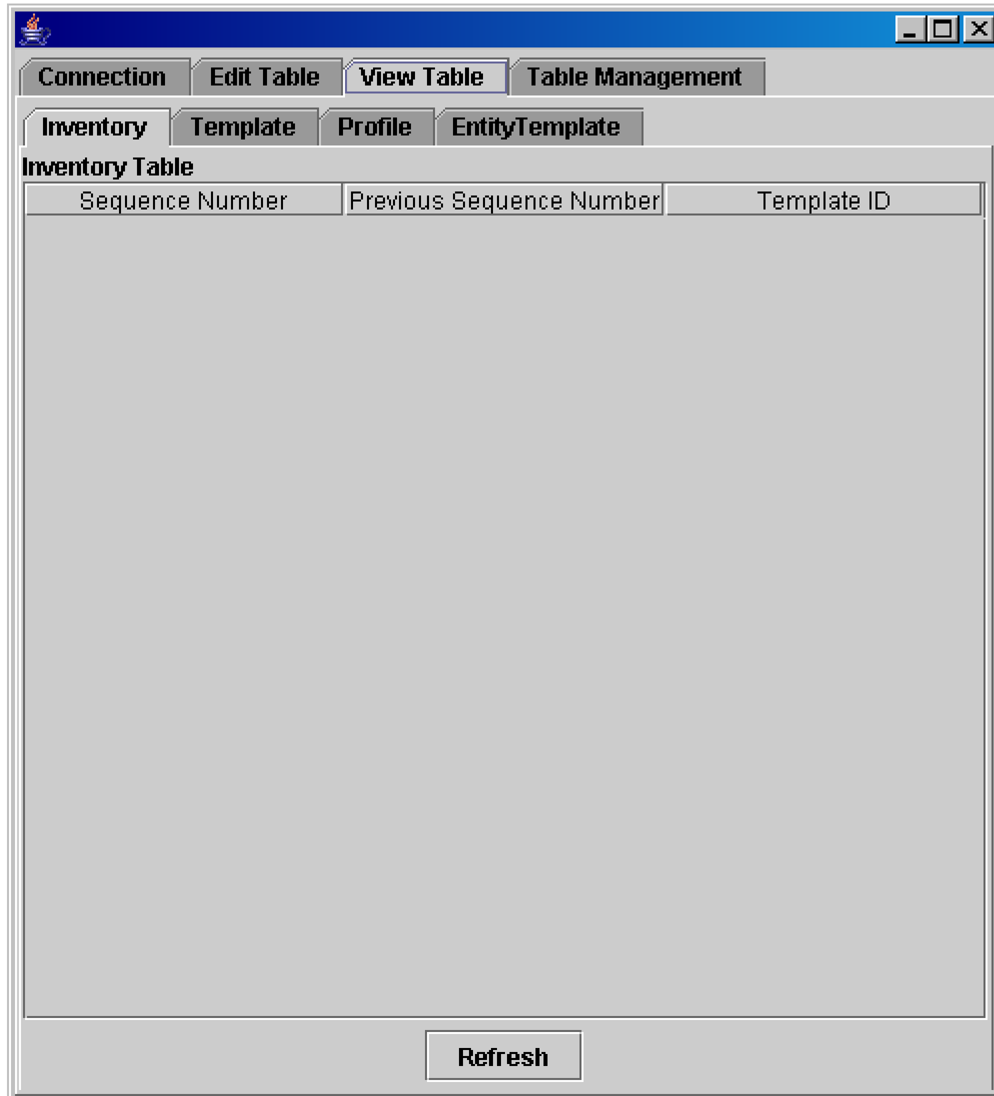


Figure 24: Viewing previously registered templates

The Figure 25 below depicts the scenario where there are no entities registered to any templates. In fact it is also possible (as can be seen by clicking the **Template** tab) that there are no registered templates. If this is the case proceed to add entries regarding the template, entity and register an entity to the template in question. To add information regarding the entities, templates etc. click on the Edit Table tab. The screen that is displayed is depicted below.

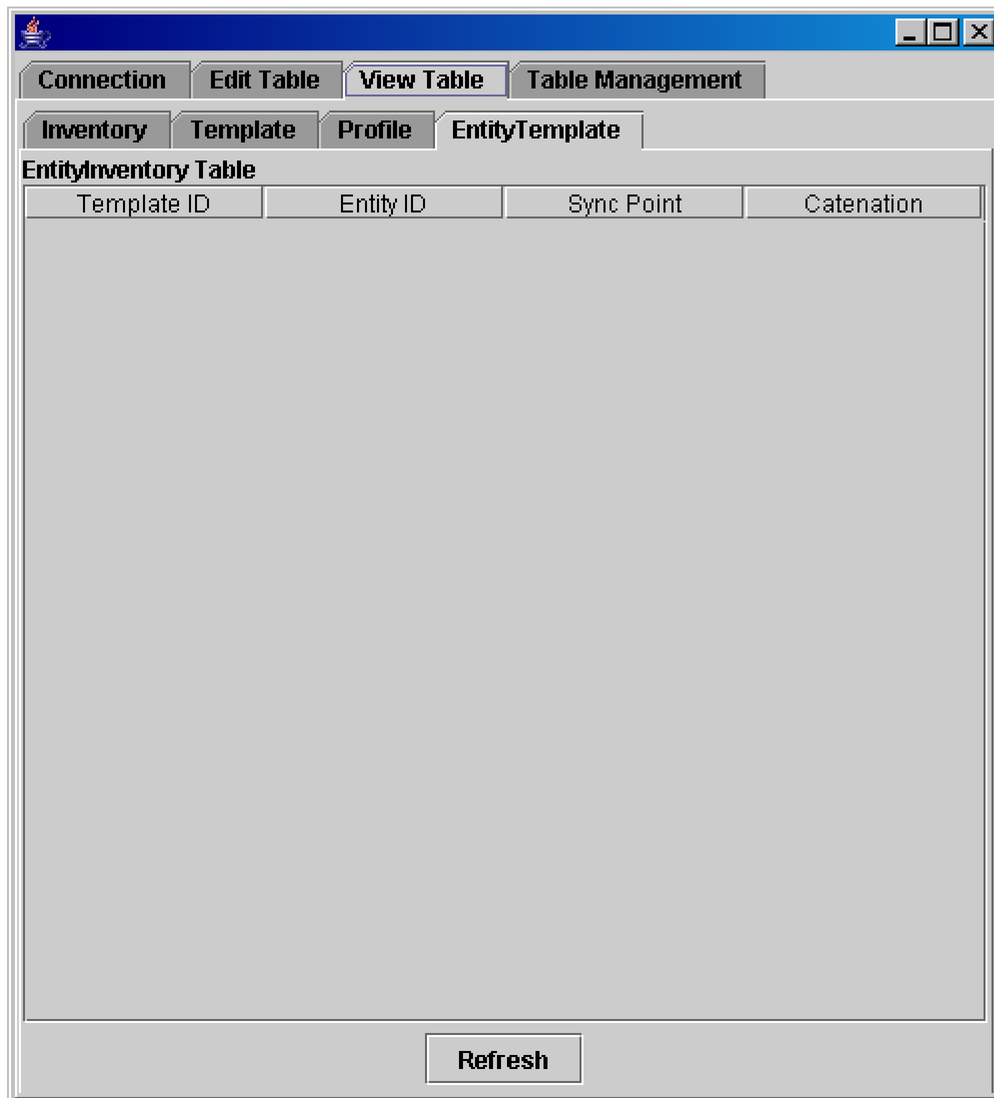


Figure 25: Viewing the templates and entities registered to these templates.

The screen that comes up when you click on the **Edit Table** tab is depicted below. First you need to add the right template in question. Proceed to click on the **Edit Template** tab.

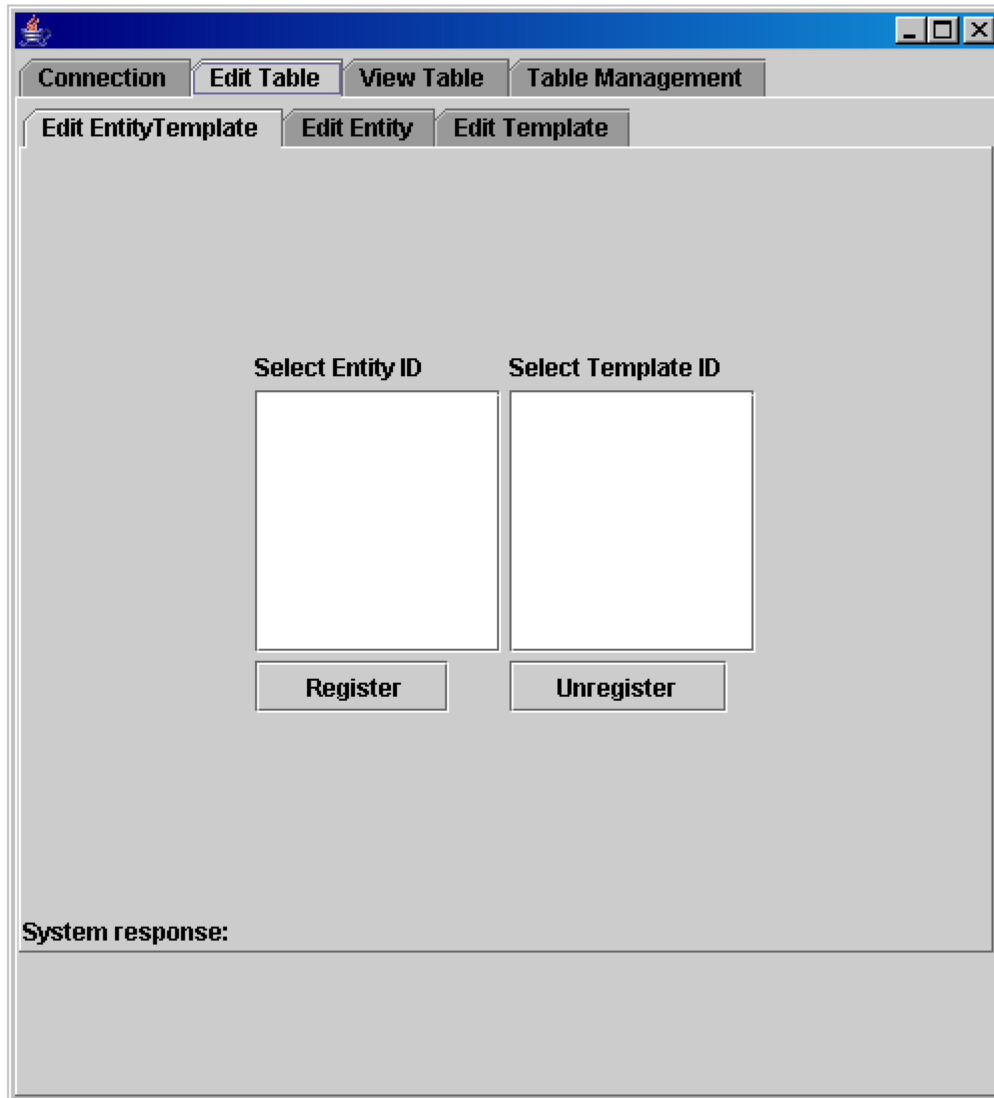


Figure 26: The Edit Table screen with sub tabs for registrations

The screen that is displayed when you click the **Edit Template** tab is depicted below. The default that is displayed is `Movies/Casablanca` with a **templateId** of 11111.

Figure 27: Registering a new template - Initial screen

Next proceed to click the **Edit Entity** Tab. The screen that is now displayed is depicted below. The display depicts the entries currently available. We are interested in adding entities with ID 4444 and 5555. If you don't see the entries in the list proceed to **register** the entities. Note that this is a one-time operation and you will not need to do this again until the database/file-system is cleared.

Figure 28: The screen for editing entity entries.

The figure below depicts the process of adding entities 4444 and 5555. Next we need to register these entities to the templateId in question. For this you need to click on the **Edit EntityTemplate** tab.

Figure 29: The screen after adding entries for entity 4444 and 5555

Next, proceed to register entities to a template by clicking on the Edit Entity Template tab. The screen below shows the registering of an entry to a template. The highlighted items indicate the entity and templateId for which registration/deregistration would be performed when the appropriate buttons are pressed.

The figure below depicts entity 4444 being registered to templateID 11111

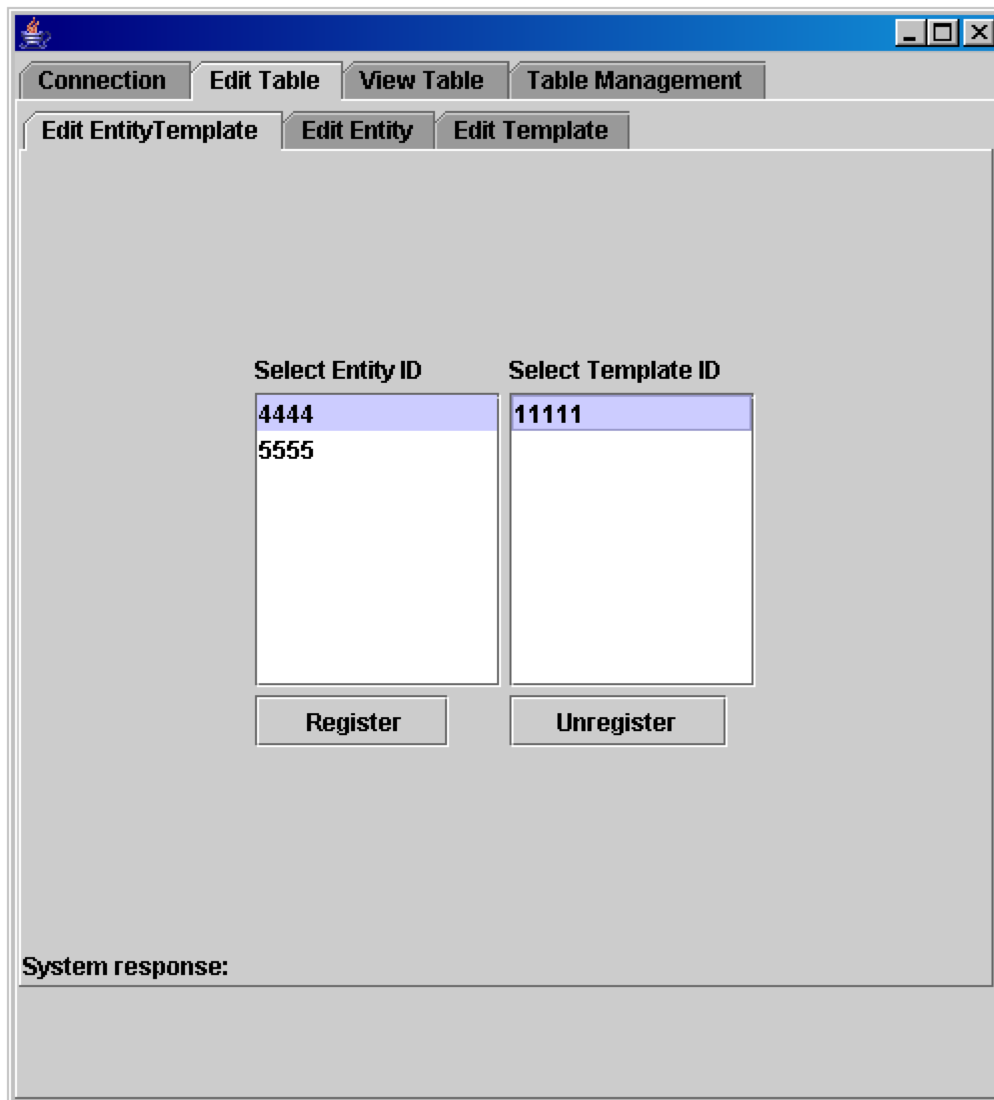


Figure 30: The screen for registering an entity to a template

The figure below depicts entity 5555 being registered to templateID 11111

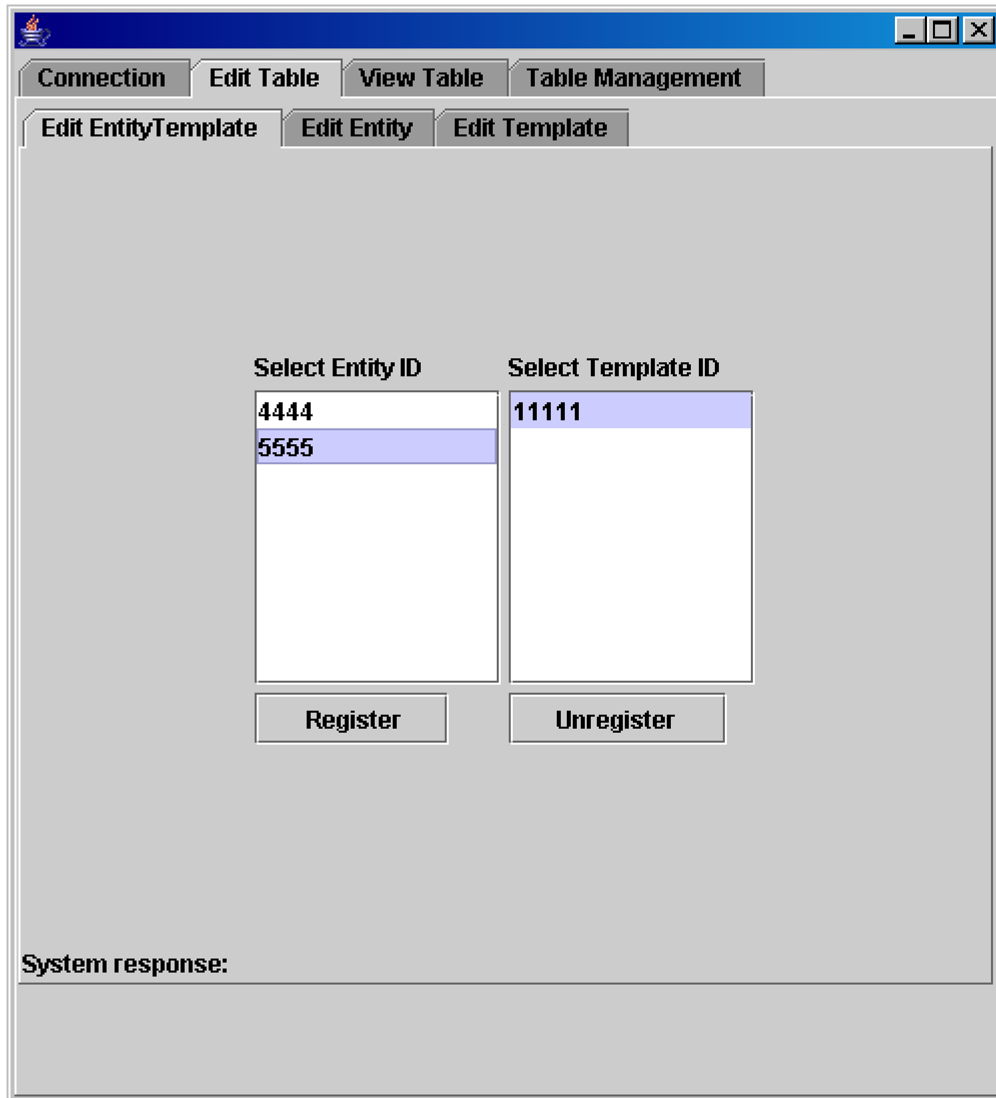


Figure 31: Another example depicting the registration of an entity to a template

Note that the registration process involving entities, templates and entities-to-templates is a one time operation for the life of the application. You need not repeat the set of instructions detailed earlier if you are returning from a scheduled downtime or a failure. The operations need to be repeated ONLY if you have explicitly deregistered the entity, template or entities-from-the-template.

The figure below shows the screen, which allows one to see the templates that were registered.

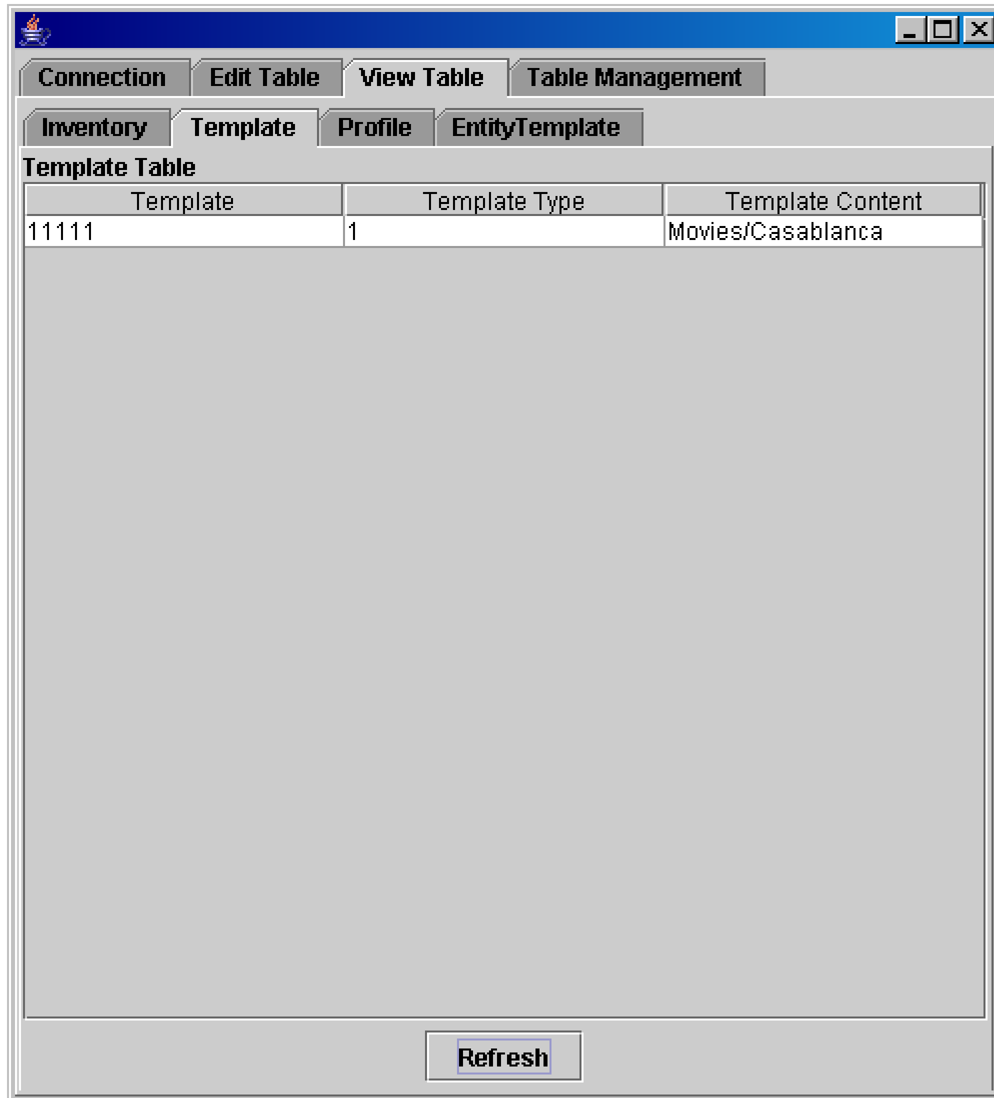


Figure 32: Viewing the registered templates

The figure below depicts the entity and the templates that they are registered to.

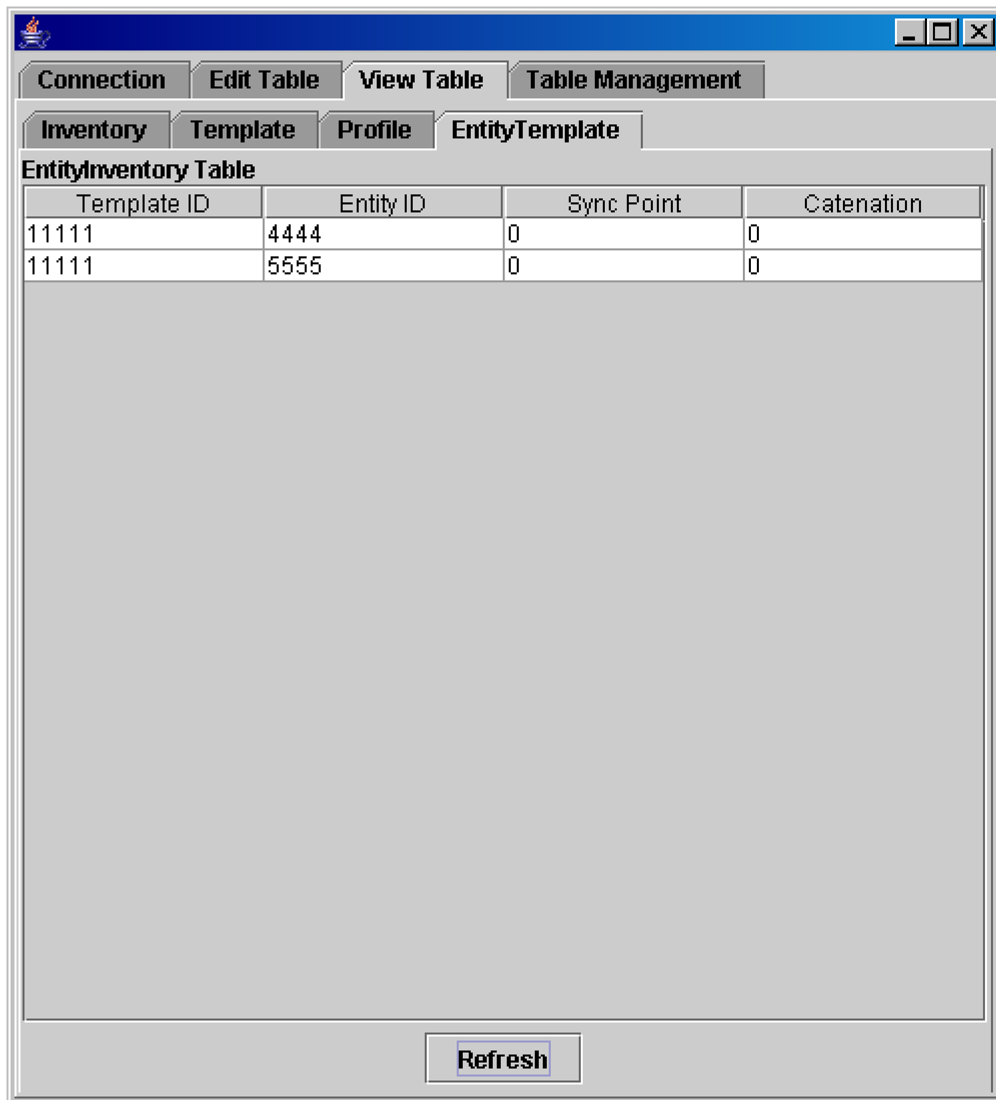


Figure 33: Viewing entities and the templates that they are registered to

7.3 The Robust Subscribers and Publishers

Now we proceed to running the robust subscriber. When you click on the **RobustSub.bat** file this is the GUI through which your actions are initiated. In the connection screen you need to enter the hostname and port number information.

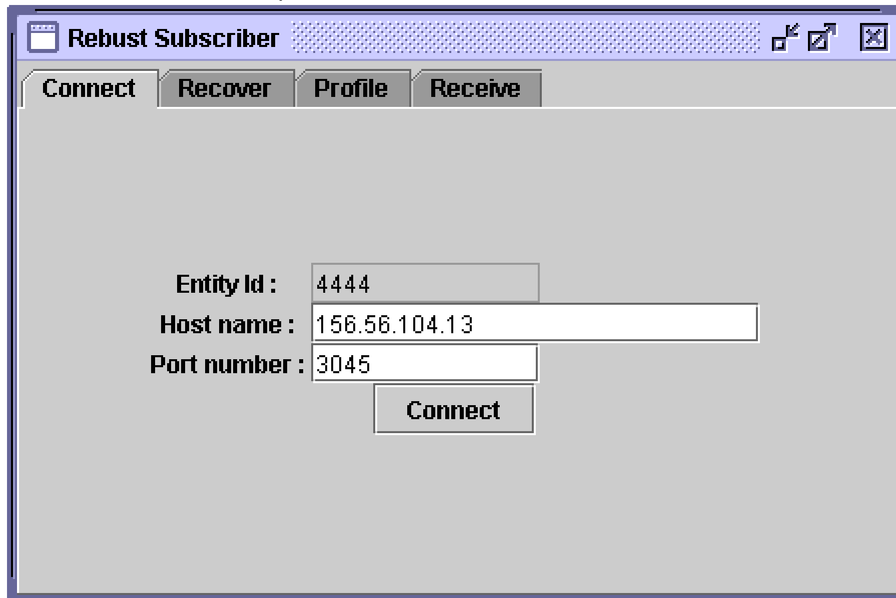


Figure 34: The robust subscriber GUI -- Connect Tab

The figure below depicts the recovery screen. You need to specify the templateId (11111) on which you are going to recover. If you had previously subscribed, you will automatically be registered to those subscriptions. However, since the first time you need to specify your subscription.

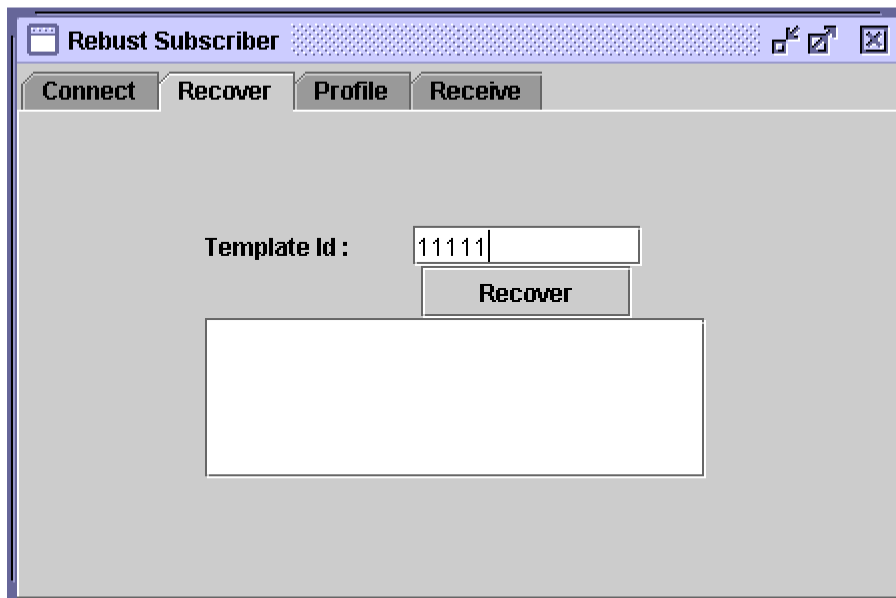


Figure 35: The robust subscriber recovery screen

The subscription profile screen is depicted below. Here you need to specify both the templateId (11111) and the subscription (Movies/Casablanca).

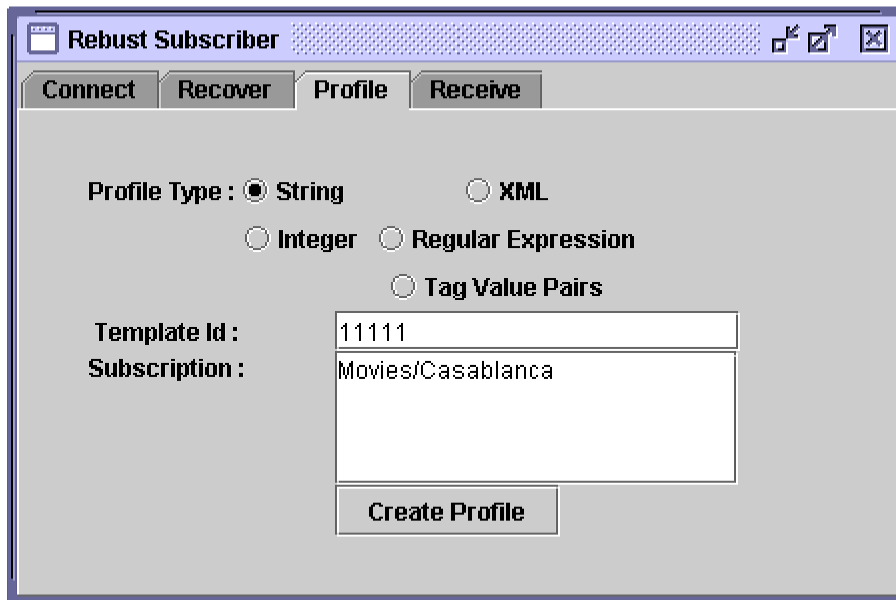


Figure 36: Robust Subscriber - Subscription screen

The figure below depicts the scenario where, if you had previously registered a subscription, you will automatically be registered to that subscription. This obviates the need to subscribe using the Profile screen.

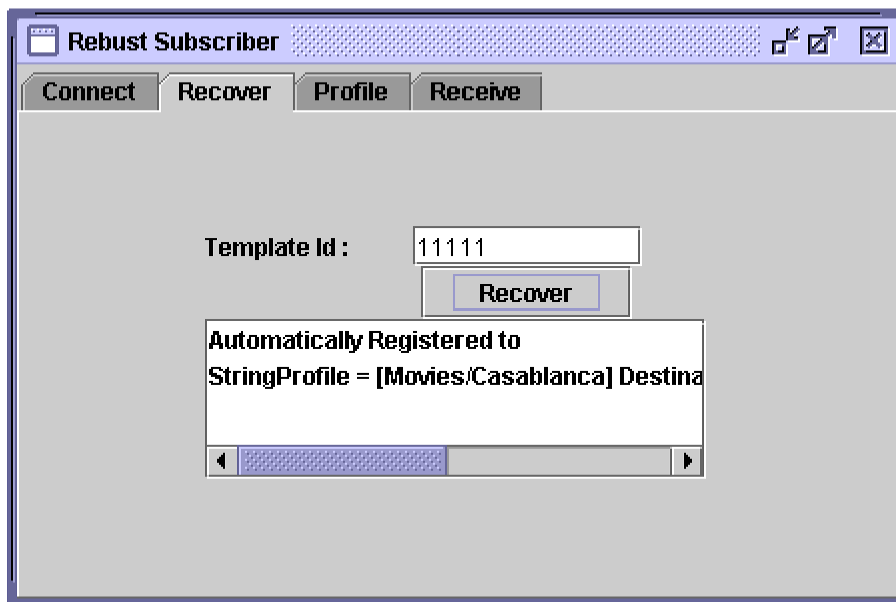


Figure 37: Robust Subscriber - Recovery screen depicting automatic subscriptions

The publisher publishing screen is similar to what's available for the robust subscriber.

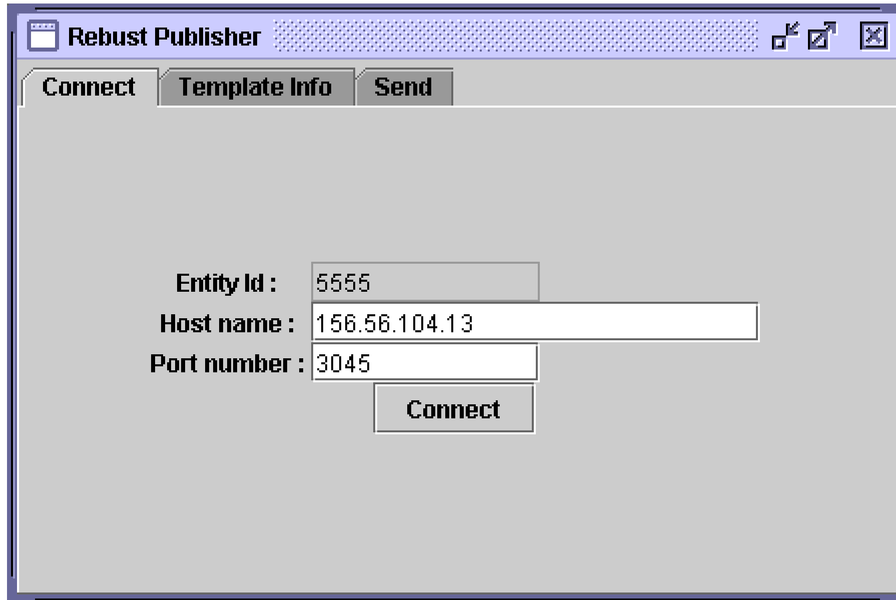


Figure 38: RobustPublisher - Connection Screen

Specify the templateId (11111) and the template (Movies/Casablanca) of the events that the robust publisher will publish.

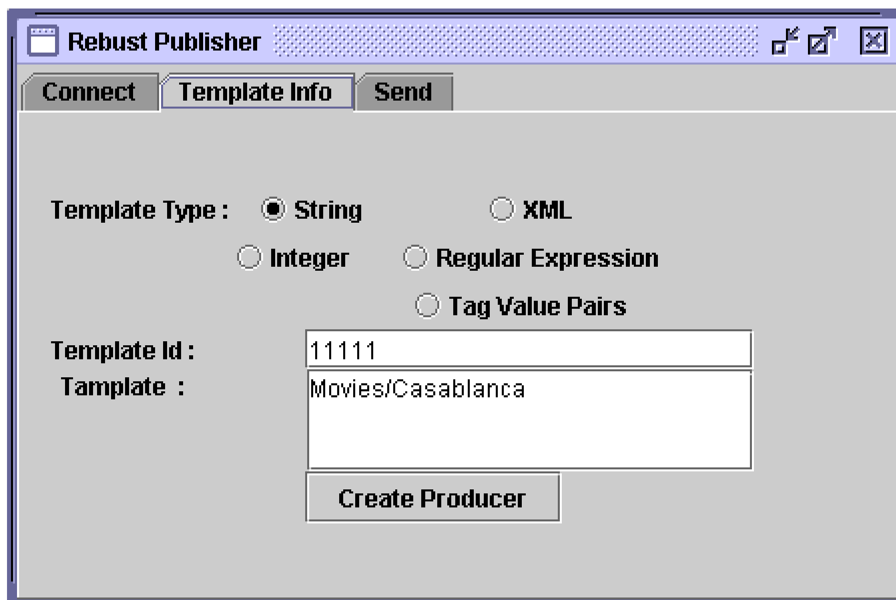


Figure 39: RobustPublisher - TemplateInfo screen

The figure below depicts the scenario where a publisher publishes a message and the robust subscriber receives it.

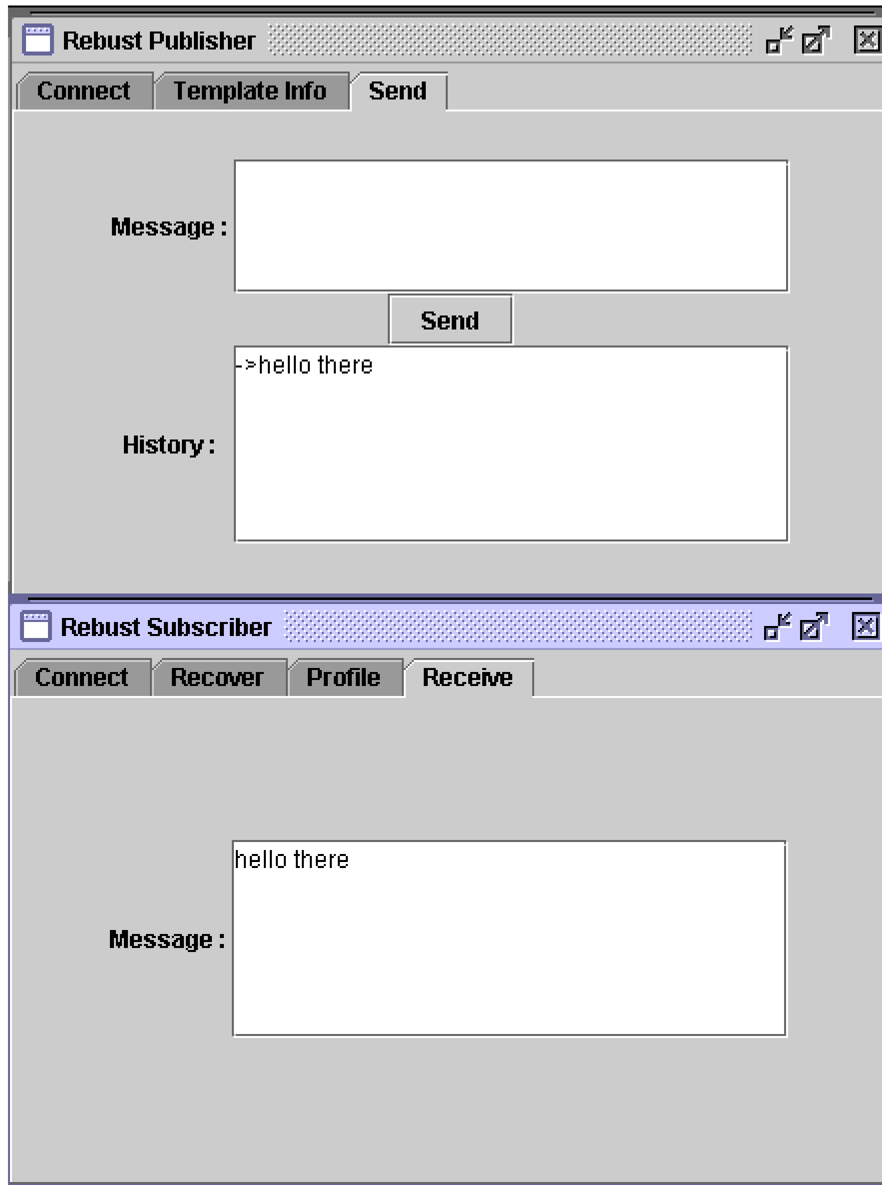


Figure 40: Publisher sending events and receiver receiving them

8 Writing JMS applications

Here we provide a brief introduction of how to develop JMS clients for NaradaBrokering. There are several excellent books and tutorials that cover developing JMS applications. Here we assume that the developer is already familiar with the developing JMS applications. NaradaBrokering provides support only for the publish-subscribe model specified in the JMS specification.

8.1 Creating a TopicConnectionFactory

The code snippet below provides an overview of the actions involved in getting a `TopicConnectionFactory`. The example below depicts the scenario when communication between the client and the broker is over TCP. The properties would be different for different transport protocols.

Table 28: Creating a TopicConnectionFactory

```
import cgl.narada.jms.*;
import javax.jms.*;
import java.util.Properties;

public class JmsApplication implements javax.jms.MessageListener {
   String hostInfo=”everest.ucs.indiana.edu”;
   Int portInfo = 3045;
   String transportType = “niotcp”;

   Properties props = new Properties();
    /** These properties pertain to setting up a TCP link */
   props.put("hostname", hostInfo);
   props.put("portnum", portInfo);

   ini = new NBJmsInitializer(props, transportType);
   /* Lookup a JMS connection factory */
   TopicConnectionFactory conFactory = (TopicConnectionFactory) ini.lookup();

   public void onMessage(Message message) {

   }
}
```

8.2 Initializing the Topic Session and Topic

The code snippet below describes the process of initializing the `TopicSession`, which requires the `TopicConnection` object created using the `TopicConnectionFactory`,

which was initialized in code-snippet outlined in Table 28. The newly initialized session is then used for the creation of a `Topic` which is the virtual channel over publishers and subscribers would communicate over.

Table 29: Creation of a TopicSession and Topic

```
TopicConnection connection =
     conFactory.createTopicConnection("guest", "password");

// Create a JMS session object
TopicSession session =
      connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

String topicString = "chat/channel/AliceAndBob"
Topic topic = session.createTopic(topicString);
```

8.3 Creating a Subscriber

The code snippet below (Table 30) depicts the creation of a `TopicSubscriber` from the `TopicSession` object. During the creation of the `TopicSubscriber` one also needs to specify the `Topic` for which the subscriber is being created.

Table 30: Creation of a Topic Subscriber

```
TopicSession subSession =
     connection.createTopicSession(false, Session.AUTO_ACKNOWLEDGE);

TopicSubscriber subscriber = subSession.createSubscriber(chatTopic);

subscriber.setMessageListener(this);


// Sample implementation of the onMessage() method, which assumes
// a TextMessage type
public void onMessage(Message message) {
     try {
         TextMessage textMessage = (TextMessage) message;
         String text = textMessage.getText();
         System.out.println(text);
     } catch (JMSException jmse) {
         jmse.printStackTrace();
     }
}
```

Additionally, to be able to consume messages published over a topic, the `TopicSubscriber` also needs to register a class that implements the

javax.jms.MessageListener. Messages over the subscribed topic are routed to the **onMessage()** method which is part of this interface and takes the javax.jms.Message as an argument. The Message class is the base message type for all messages supported within the JMS specification. The next section describes the process of creating JMS messages.

8.4 Message Types

The JMS specification incorporates support for several different message types, each of which has a set of methods to manipulate the message types. The message types supported by the JMS specification include: BytesMessage, TextMessage, MapMessage, ObjectMessage and StreamMessage. The snippet below demonstrates the creation of TextMessage using the TopicSession object.

Table 31: Creation of JMS Messages

```
TextMessage message = pubSession.createTextMessage();

//Manipulating the message type. This would vary for different message types
message.setText("[" + userName + "] : " + text);
```

8.5 Creating a Publisher

The code snippet below (Table 32) demonstrates the creation of the TopicPublisher. The process is quite similar to the creation of a TopicSubscriber which we outlined earlier in section 8.3. The TopicSession object is needed to create the TopicPublisher to a specific Topic which is specified in the argument. The snippet also outlines the process of publishing messages using the publisher.

Table 32: Creation of Publisher and publishing messages

```
TopicPublisher publisher = pubSession.createPublisher(jmsTopic);

TextMessage message = pubSession.createTextMessage();
message.setText("[" + userName + "] : " + text);

publisher.publish(message);
```

8.6 Running the sample JMS chat application

There is a simple JMS chat application included in the distribution. Let us say that the broker process is running on **machine.ucs.indiana.edu** on the non-blocking TCP port of **3045**. Further, say that the Chat user's identity is **tom**. The command to run the Chat application would be –

```
java cgl.narada.samples.Chat machine.ucs.indiana.edu 3045 tom
```

8.7 Unsubscribing Topics

In the JMS specification (1.0.2b) the unsubscribe operation is only specified for *durable subscriptions* that have been assigned a *name*. Surprisingly, there isn't an unsubscribe operation associated with a **TopicSubscriber**. The **unsubscribe()** operation has to be invoked on the **TopicSession**, and as mentioned previously available only for durable subscriptions.

What we have done in NaradaBrokering's JMS support is to provide **unsubscribe()** support for all subscriptions. Furthermore, if there are multiple sessions that have subscribed to the same topic, unsubscribing on one of these sessions will not affect the subscriptions in the other sessions. However, if you subscribe to the same topic from the same session one of the subscribers will be inactivated, though you cannot know which one, since there is no way to distinguish them. If however, you do need to unsubscribe a specific subscriber we have incorporated support for this. You will need to do **unsubscribe()** after casting the subscriber to **cgl.narada.jms.JmsTopicSubscriber**.

9 Broker Discovery

In this chapter we provide information on automating discovery of brokers.

9.1 Discovering Brokers

- Start the Broker Discovery Node. This step is not mandatory but can be more useful in disseminating broker discovery requests. Other approach is to use multicast. Refer to the paper for more information on the disadvantages and issues with using multicast.
- The NB package comes with a precompiled WAR (Web Archive for deploying in a servlet container such as tomcat). Simply drop the BDN.war file in the webapps/ directory and restart the server. This should start the broker discovery node. To check, go to http://yourHost:serverPort/BDN.
- The brokers themselves need to be told the location of the broker discovery node. This can be done by including the location of the BDN register service in the NB_HOME/config/BrokerConfiguration.txt file. For e.g., if the BDN.war was deployed on localhost at port 8080 then the BDNList entry in the BrokerConfiguration.txt file would look like BDNList=http://localhost:8080/BDN/servlet/BDN.

  Similarly the URL for discovery needs to go into the ServiceConfiguration.txt file. This usually looks like BDNDiscoveryList=http://localhost:8080/BDN/servlet/Discover.
  Replace localhost with the host / ip address of the machine on which the BDN runs and 8080 with the port on which it runs!

9.2 Using the Broker Discovery Helper

Broker Discovery Helper is a utility to help locate brokers programmatically. Available brokers may be located as follows:

```
int eid = 5000; // some randomly generated unique ID

// timeout to wait for discovery responses (in milli sec)
int timeout = 5000;

// Maximum responses to wait. IF we get these many responses before the
// timeout occurs, we simply disregard all other responses.
int maxResponses = 3;

// The broker target set size. (maxSetSize <= maxResponses)
int maxSetSize = 2;
```

```
// Assume NB_HOME was set or hardcode it here...
// OR better still pass it as a command line parameter.
// Basically some way to identify the location of the Service
// Configuration file.
String configPath = NB_HOME + "/config/ServiceConfiguration.txt";

BrokerDiscoveryHelper bdh = new BrokerDiscoveryHelper(eid, configPath,
               timeout, maxResponses, maxSetSize);

// Only brokers which have the following protocol link services enabled must
respond.
String[] protocolSet = { "tcp", "niotcp"};

BrokerDiscoveryResponse[] responses = bdh.discover(protocolSet, "",
               networks);

for(int i = 0; i< responses.length; i++ ) {
         System.out.println("Response: " + i + "\n" +
               responses[i].toString());
}

// This returns the final broker to use, using the default "Best broker
// selection" algorithm. Alternatively, simply write your own procedure //
for selecting the best broker from the above array
// of broker responses
return bdh.selectBestBroker(responses);
```

10 Topic Creation & Discovery

Please refer to the paper for more information on the Topic Creation & Discovery scheme.

10.1 Topic Creation

10.1.1 Starting the Topic Discovery Node

The Topic discovery node (TDN) has been implemented as a NB client. To create a topic, a TDN must be present and connected to the brokering network. A TDN may be started using the following command

```
java -classpath %CLASSPATH%;%NB_CP%
        cgl.narada.discovery.topics.TopicDiscoveryNode
```

This command also takes optional parameters, namely <BROKER_HOST> <BROKER_PORT> <PROTOCOL_TO_USE>

Default values are localhost, 25000, niotcp !
Refer to Javadocs for the most updated information on command line parameters.
This step is mandatory.

10.1.2 Creating Topics

The NaradaBrokering packages contain a utility class `cgl.narada.discovery.topics.Entity` that discovers a TDN and works with the selected TDN to arrive at a Topic Advertisement. However first, the process requires the user's X.509 Certificate, Private key and the Root CA's public key. This may be loaded using the procedure outlined here.

Once these values have been loaded, we can create a topic using the following code.

```
int eid = 5000; // some randomly generated unique ID

// Name of the topic to create (in this case a string topic)
String topicName = "/sports/NBA";

Entity e = new Entity(eid,
        // Path to the Service configuration file
        "/path_to/ServiceConfiguration.txt",

        // Alias of the user who is creating the topic
```

```
        // (called as topic owner)
        "testuser",

        cert, // Certificate of the test user
        priv, // Private Key of the test user
        rootCA, // Public Key of the ROOT CA

        // Hostname / IP Address of the broker to which a
        // connection must be made
        "host",

        "port", // Port on which the broker is accepting
                // connection, NOTE: this is a string parameter

        "protocol" // The communication protocol to use
                   // (E.g. tcp, udp, niotcp etc...)
);

if (e == null) {
        // CHECK... e must not be NULL...
        System.out.println("ERRRORRRR !!!!");
        return;
}

if (e.sendTDNDiscoveryRequest(10000)) {
        System.out.println("Found TDN ! Proceeding to create TOPIC !!");

        // Set the topic validity
        Calendar until = Calendar.getInstance(
                TimeZone.getTimeZone("GMT-5"));
        until.add(Calendar.HOUR, 1); // E.g. Valid for 1 hr. from now

        // Refer cgl.narada.event.TemplateProfileAndSynopsisTypes for
        // different types of topics that can be created.
        // Currently String, REGEX, Integer are supported

        if (!e.createTopic("SELF", until, "Test Topic",
                TemplateProfileAndSynopsisTypes.STRING,
                topicName, 5000)) {
                System.err.println("Could not create topic ! "
                                + "Aborting...");
                return false;
        }

        System.out.println("TOPIC Created: UUID -> " +
                                e.getTopicUUID(topicName));
        return true;
```

```
} else {
        System.out.println("NO TDN found within specified "
                + "timeout period !!");
        return false;
}

// Get Signed Topic Ad
SignedTopicAdvertisement sta =
                e.getSignedTopicAdvertisement(topicName);
```

10.2 Topic Discovery

10.2.1 Discovering Topics

Once a topic has been created, it may be discovered using the right credentials. Current scheme allows discovery for any client who presents a valid X.509 certificate. Restriction on topic usage (publish / subscribe) is addressed in the security framework. Once a topic has been created, any topic discovery requests automatically check for expired topics. If an expired topic is found it is removed. A topic discovery may be done as follows

```
int eid = 5000; // some randomly generated unique ID

// If a specific TDN is to be used, then use this id, else keep it null
// If NULL, the first TDN to respond will be used.
String tdnID = null;

// Name of the topic to create (in this case a string topic)
String topicName = "/sports/NBA";

// For different types, refer cgl.narada.discovery.topics.Topics
int matchingType = Topics.MATCHING_STRING;

// Maximum number of responses to gather until a timout of 5 seconds.
int maxTopics = 2;

TopicDiscoveryClient tdc = new TopicDiscoveryClient(
        eid, // Entity Id to use when connecting to the broker

        // Path to the Service configuration file
        "/path_to/ServiceConfiguration.txt",

        // Certificate of the user trying to discover mathcing topics
        cert,
```

```
        // Private Key of the user trying to discover mathcing topics
        priv,

        // Hostname / IP Address of the broker to which a
        // connection must be made
        "host",

        // Port on which the broker is accepting connection,
        // NOTE: this is a string parameter
        "port",

        // The communication protocol to use
        // (E.g. tcp, udp, niotcp etc...)
        "protocol"
);

// Get Signed Topic Ad
SignedTopicAdvertisement[] stas = tdc.discover(matchingType,
                                      topicName, tdnId, maxTopics);
```

Once a list of signed topic advertisements is received, the client may pick one to decide the topic on which he wishes to communicate / listen to events.

11 Root Provider

Root provider is the certificate provider used to issue digitally signed certificates. It performs the following functions

- Issue digitally signed certificates
- Store certificates in the key-store
- Delete certificates (X.509 certificate, public key and private key) from the key-store

Root provider can be used to create users for use in topic creation and discovery and security framework.

11.1 Using the Root Provider

Package: `cgl.narada.service.security.securityProvider`
Examples:
Generating ROOT Certificate (to be done only once)

```
CertificateManager certMan = new CertificateManager();

// To use default password use null.
// OR specify a different password in the second parameter
certMan.init(
     "/home/hgadgil/tmp/narda/keystore/NBSecurityKeys.keys",
     Null
);

ROOTSecurityProvider.GenerateRootCertificate(certMan);
```

Certificate manager stores the most commonly used key-store properties for certificate management, particularly the key-store type, key-store provider, key algorithm, key-store password. Typically these values are default, however other values may be used by using the `CertificateManager(Properties)` constructor. The `java.util.Properties` can contain the various properties to use. The following table lists all the possible properties.

Table 33: Properties that can be specified for the constructor

Property	Used for	Default Value
KEYSTORE_PATH	Location of the keystore	No default. MUST be specified during init
ROOT_CA_ALIAS	Alias used for the root's certificate and keys	rootca
KEYSTORE_TYPE	Type of the keystore	JKS
KEYSTORE_PROVIDER	Keystore provider	SUN
KEY_ALGORITHM	Algorithm for key generation	RSA
KEYSTORE_PASSWORD	Password to access the keystore	passpass

To issue certificates, the ROOT Provider creates a RSA key, gets the CSR (Certificate signing request) and digitally signs it using the ROOT provider's private key to create a certificate for the client. This process is illustrated below

```
CertificateManager certMan = new CertificateManager();

// To use default password use null.
// OR specify a different password in the second parameter
certMan.init(
     "/home/hgadgil/tmp/narda/keystore/NBSecurityKeys.keys",
     Null
);

// Parameter 1: Specifies the alias to use
// Parameter 2: Specifies the DN of the user for whom the digital
//              certificate is being issued
// Parameter 3: Specifies the validity of the certificate (in days)

ROOTSecurityProvider.IssueSignedCertificate("testuser",
  "\"CN=TEST-USER,OU=OrgUnit,O=Organization,L=Location,C=country",
  50
);
```

1. Alternatively if a certificate is issued, this may be requested using the Java's Keytool command. This is usually located in `<JAVA_SDK_HOME>/bin/keytool`. Refer to Java SDK for using Keytool. This may be found at http://java.sun.com/j2se/1.4.2/docs/tooldocs/windows/keytool.html!

2. To delete a certificate for an alias "testuser", use the following

```
CertificateManager certMan = new CertificateManager();

// To use default password use null.
// OR specify a different password in the second parameter
certMan.init(
     "/home/hgadgil/tmp/narda/keystore/NBSecurityKeys.keys",
     Null
);

// Parameter 1: Specifies the alias of the user whose certificate
// (key, certificate and entry in keystore) are to be deleted
ROOTSecurityProvider.DeleteCertificate("testuser")
```

11.2 Loading Certificates and Keys

Once the certificates have been created, they may be loaded using the key management implementation in NaradaBrokering. The newly added topic creation/discovery [Section 10] and security framework [Section 12] heavily use the digital certificates and keys. To load the keys and certificate for a user "testuser", use the following code

```
CertificateManager certMan = new CertificateManager();

// To use default password use null.
// OR specify a different password in the second parameter
certMan.init(
        "/home/hgadgil/tmp/narda/keystore/NBSecurityKeys.keys",
        Null
);

Certificate myX509Certificate = CertificateUtil.getCertificate(
        certMan, "testuser");

PrivateKey myPrivateKey = CertificateUtil.getPrivateKey(
        certMan, "testuser");

// Frequently to validate a certificate, one also needs the root's
// public key. This may be loaded as follows
PublicKey rootCAPublicKey = CertificateUtil.getPublicKey(
        certMan, certMan.ROOT_CA_ALIAS);
```

12 Security Framework

Please refer to the NaradaBrokering security paper for more information on the Security Framework.

12.1 Creating Security Tokens and securing topics

12.1.1 Starting the Key Management Center

The Key Management Center (KMC) may be started using the following command

```
java -classpath %CLASSPATH%;%NB_CP%
        cgl.narada.service.security.kmc.KMCService
```

This command also takes optional parameters, namely <BROKER_HOST> <BROKER_PORT> <PROTOCOL_TO_USE>. The default values are localhost, 25000, niotcp respectively. This step is mandatory.
Please refer to Javadocs for the most updated information on command line parameters.

12.2 Creating Secure Topics

To create a secure topic, a TDN and a KMC must be present and connected to the brokering network. To register a secure topic, first step is to create a topic using the Topic Creation mechanism described earlier. The NaradaBrokering package contains a utility class `cgl.narada.service.security.kmc.KMCClient` that aids in the process of creating a security token for a given topic. However first, the process requires the user's X.509 Certificate, Private key and the Root CA's public key. This may be loaded using the procedure outlined in Section 11.2.

Once these values have been loaded, we can create a topic using the following code.

```
// Signed topic advertisement as obtained during the
// topic creation process
SignedTopicAdvertisement sta = ...;

KMCClient client = new KMCClient(
        cert, // Certificate of the test user
        priv, // Private Key of the test user
        rootCA, // Public Key of the ROOT CA
```

```
        "/client/c1", // Name of the topic on which the KMC replies back

        // Path to the Service configuration file
        "/path_to/ServiceConfiguration.txt",

        // Hostname / IP Address of the broker to which a
        // connection must be made
        "host",

        // Port on which the broker is accepting connection,
        // NOTE: this is a string parameter
        "port",

        // The communication protocol to use
        // (E.g. tcp, udp, niotcp etc...)
        "protocol"
);

// Set the topic validity
Calendar until = Calendar.getInstance(TimeZone.getTimeZone("GMT-5"));
until.add(Calendar.HOUR, 1); // E.g. Valid for 1 hr. from now

// Set the access control list for publish subscribe...
Hashtable pubs = new Hashtable();


// NOTE: Currently all subscribers and publishers *MUST* be specified
// This includes the topic owner too. This would be automated in the
// next version

// Important step...
pubs.put(((X509Certificate) c1_cert).getSubjectDN().getName(), until);

Hashtable subs = new Hashtable();

// Important step…
subs.put(((X509Certificate) c1_cert).getSubjectDN().getName(), until);

// This client has the subscribe right untile the time
// specified by 'until'
subs.put(
        // DN of the client who has been given right
        "CN=client2, OU=CGL, O=IU, L=Bloomington, C=US",

        // Time until this client has this right
        Until
```

```
);

// Ok, now register the topic and get a security token
SecureTopicKeyResponse resp = client.registerTopic(
        pubs, // set publishing rights
        subs, // set subscribing rights
        sta,  // the signed topic advertisement
        cert,  // Topic owner's X.509 certificate
        until, // Validity of secure topic
        algo,   // Algorithm of secret key generation (Default AES)
        keylen, // Key length of the secret key (default 192 bits)

        // time for which to wait for a response from the KMC
        // (in milli seconds)
        5000
);

// Retrieve the signed security token...
SignedSecurityToken token = resp.getSignedSecurityToken();
```

12.3 Signed Security Token Retrieval

Once a secure topic has been registered with a KMC and a secret key created, allowed publishers / subscribers may retrieve the token. The first step is to get a signed topic advertisement for which a security token is desired. This is done using the procedure outlined in section 10.

```
// Signed topic advertisement as obtained during
// the topic discovery process
SignedTopicAdvertisement sta = ...;


KMCClient client = new KMCClient(
        cert, // Certificate of the test user
        priv, // Private Key of the test user
        rootCA, // Public Key of the ROOT CA
        "/client/c1", // Name of the topic on which the KMC replies back

        // Path to the Service configuration file
        "/path_to/ServiceConfiguration.txt",

        // Hostname / IP Address of the broker to which
        // a connection must be made
        "host",
```

```
        // Port on which the broker is accepting connection,
        // NOTE: this is a string parameter
        "port",

        // The communication protocol to use
        // (E.g. tcp, udp, niotcp etc...)
        "protocol"
);

// Specify the requested rights... if these rights match the ones
// specified by the topic owner, only then is a security token
// issued...
TopicRights requestedRights = new
         TopicRights(TopicRights.SUBSCRIBE_RIGHT);

SecureTopicKeyResponse resp = client.requestTopicKey(
        // The topic synopsis from the signed topic ad.
        sta.getTopicAd().getTopicSynopsis(),

        cert, // Requestor's X.509 certificate
        requestedRights, // Requested rights
        5000 // timeout
);

if (resp == null)
        System.out.println("Request Denied / Timeout occurred !");
else
        System.out.println("Token recieved... !");
```

Once a list of signed topic advertisements is received, the client may pick one to decide the topic on which he wishes to communicate / listen to events.

12.4 Secure Publishing of events

A secure event may be published by setting appropriate flags in the `ProducerConstraints`. Along with this, one also needs to add the security token obtained from the KMC. Note that one always publishes to the topicUUID (modified topic synopsis) obtained in the topic discovery process.

```
// Signed topic advertisement as obtained
// during the topic discovery process
SignedTopicAdvertisement sta = ...;

String topicName = sta.getTopicAd().getTopicSynopsis();
```

```
TemplateInfo ti = new TemplateInfoImpl(
        12345, // Id representing the template

        // The type of the tempalte (used while matching)
        TemplateProfileAndSynopsis.STRING,

        topicName // Modified synopsis as obtained above...
);

EventProducer producer = ... ; // Previously created...
ProducerConstraints pc = producer.createProducerConstraints(ti);

pc.setSendSecurely();

// This is optional and defaults used are
// algorithm = SHA1withRSA
// mode = CBC
// padding = PKCS7Padding

// Used for finer control over the digital signature process
Properties props = new Properties();
props.put(ProducerConstraints.SIGNING_ALGORITHM, "SHA1withRSA");
props.put(ProducerConstraints.CIPHER_MODE, "CBC");
props.put(ProducerConstraints.CIPHER_PADDING, "PKCS7Padding");

SecureTopicKeyResponse resp = ... // previously obtained

pc.setSecurityToken(
        // Signed security token (contains rights signed by the KMC)
        resp.getSignedSecurityToken(),

        resp.getKey(), // Secret key for payload encryption
        props // For digital signature
);

NBEvent event =  ...// create the event

// When actual publishing, include the producer constraints...
producer.publishEvent(nbEvent, pc);
```

12.5 Receiving Secure Events

Secure event may be received by setting including the security token in the topic subscription request.

```
// Signed topic advertisement as obtained during the
// topic discovery process
SignedTopicAdvertisement sta = ...;
String topicName = sta.getTopicAd().getTopicSynopsis();
EventConsumer consumer = ... ; // Previously created...
int entityId = ...; // Some integer identifying this entity...
Profile profile = ...; // Profile creation

SecureTopicKeyResponse resp = ... // previously obtained

// If encrypted payload is to be delivered then set this to true,
// else false
boolean doNotDecryptPayloadBeforeDelivery = false;

// Create consumer constraints
ConsumerConstraints cc = consumer.createConsumerConstraints(profile);
cc.setReceiveSecurely(entityId);

cc.setSecurityToken(

        // Security token identifying rights (signed by KMC)
        resp.getSignedSecurityToken(),

        resp.getKey(), // Secret key for payload decryption
        props, // Currently unused, For future use
        doNotDecryptPayloadBeforeDelivery
);

// Include the consumer constraints in subscription
consumer.subscribeTo(profile, cc);
```

13 The C++ Bridge for NaradaBrokering

In this chapter we describe the C++ bridge for NaradaBrokering. We have two different approaches to the C++ bridge. The first one is Sockets based, while the second one is JNI based.

13.1 C++ Socket Client for Naradabrokering

The first section of this user guide will take you through the installation process and the next section shows how to use the simple chat client. The final section explains the architecture and how to utilize the C++ Client to implement communication channels.

13.1.1 Configuration

13.1.1.1 Broker Configuration

> Note: The current implementation of the C++ Client works on **Intel**-based architectures. The differences in the **endianness** of various architectures require different compilations. More explanation about this will follow in section three.

Download and unzip the Naradabrokering from http://www.naradabrokering.org/ to some local directory (say **NB_HOME**)

Start the Broker using the **startbr.sh** shell scripts in the bin directory inside **NB_HOME.**

> Note: If you need to handle larger payloads, please change the line in the startbr.sh

```
java -classpath $cp cgl.narada.node.BrokerNode $brokerConfigFile
$serviceConfigFile $brokerCommunicatorPort&
```

> to

```
java -Xmx<max value>m -Xms<min value>m -classpath $cp
cgl.narada.node.BrokerNode $brokerConfigFile $serviceConfigFile
$brokerCommunicatorPort&
```

Use the BrokerConfiguration.txt found in the **config** directory inside **NB_HOME** to change the ports that the broker used for communication. Please note that this step is not mandatory, using the default ports is fine.

13.1.1.2 Compiling the C++ Client

Download and unzip the **nbcpp.tar.gz** to a local directory (say **CLIENT_HOME**)

Inside **CLIENT_HOME** you will find a **src** directory, which contains the C++ code.

Compile the simple chat client using the *make* tool. Use the following command.

```
make pubsub
```

1. This will create an executable **pubsub** in the **src** directory itself.

13.1.2 Simple Chat Client

Once the C++ Client is compiled go to **CLIENT_HOME/src** directory and run **pubsub** to start one chat client. This will require few input parameters as explained below.

```
./pubsub 7799 44567 /test/topic 127.0.0.1 5045
```

The first integer argument is the **entityId**, which identifies this client in a given broker network.
The, next integer argument is the **templateId** which is a unique integer associated with a given topic.
The third parameter is the **topic** for which this client publishes and subscribes. This can be any string without intemediate spaces.
The fourth and the fifth arguments are the host address and the port number of the broker. Please note that we need to use the TCP port of the broker. This would be port 5045 if you are using the default port numbers.

Once the chat client is started, start another chat client with different **entityId**. The next step is to see the Chat program in action by typing in few messages.

To exit from the chat client type **$<return>.**

13.1.3 The Architecture

The C++ Client establishes a TCP connection with a given broker and supports exchanging of pub/sub messages. The following diagram shows the architecture of the C++ Client.

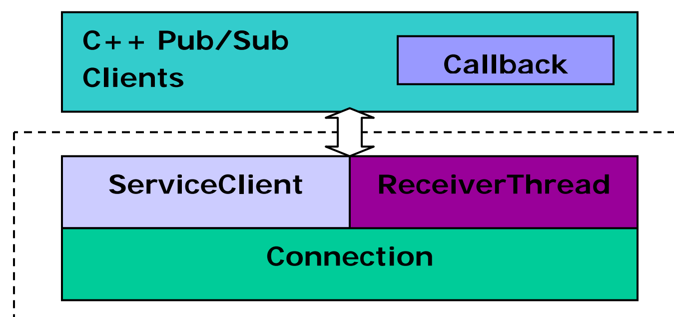


Figure1: Architecture of the C++ Client for NaradaBrokering.

The API that the C++ programmer needs to work with has one class - **ServiceClient** and the **Callback** interface. The **ServiceClient** hides the rest of the components shown in the above architecture diagram from the user, and provides four basic methods that give a publish/subscribe interface for the C++ clients. These methods are listed below:

```
void init(string host,int port,int entityId,int templateId);
void subscribe(string topic,Callback *callback);
void publish(string topic,char* bytes,int length);
void close();
```

First, the client needs to establish a connection using the **init(..)** method shown above which takes four input parameters.

Host	host address of the broker
Port	TCP port of the broker(default is 5045)
entityId	This will identify the client uniquely in a broker network
templateId	A template Id for this connection

If the client needs to subscribe to a specific topic, then the method to use is: **subscribe(..)**. This method takes a topic and a Callback object as input parameters.

topic	String parameter, which specifies the topic to which the client needs to subscribe.
callback	This should be an implementation of the Callback interface provide by the C++ API. The client is expected to implement the onEvent(NBEvent *nbEvent) method of the callback. The C++ API will call this method for any event received for a topic that this client is subscribed.

To publish messages to a topic, the client can utilize the **publish(..)** method. This method takes three parameters as explained below.

topic	String parameter which specifies the topic to which the client needs to publish events.
Bytes	This is the content payload of the message and can be any number of bytes
length	Length of the byte array of the content payload

Finally if the client needs to close the broker connection, then it can use the **close()** method of the **ServiceClient**.

13.1.4 Issues specific to Endianness

The current implementation supports Intel-based machines that use Little Endian ordering when storing bytes. This affects the way we store multi-byte data types and send them over the communication channels. Java handle bytes in BigEndian format as inherited from its Solaris roots. However, the Intel based architectures use LittleEndian format, and hence a conversion is required when exchanging messages between these architectures. The current implementation assumes a 32-bit value for integers and 16-bit values for short data types. This part requires little more research to make it generic for both 32-bit and 64-bit architectures. However, this difference does not affect the usage since the C++ client accepts a byte array as the content payload which is unique across the above platforms.

13.1.5 Simple Pub/Sub Example

The following code fragment shows the methods that need to be used in order to write a pub/sub client using the above API.

```
ServiceClient serviceClient;
/*Establishes a connection*
serviceClient.init(host,port,entityId,templateId);

MyCallback callback;
/*Subscribed to a topic*/
serviceClient.subscribe(contentSynopsis,&callback);

/*Publishes a message*/
serviceClient.publish(contentSynopsis,msg,strlen(msg));

/*Close the connection*/
serviceClient.close();
```

13.2 C++ Bridge for NaradaBrokering (JNI-based)

The first section of this user guide will take you through the installation process and the next section shows how to use the simple chat client. Final section explains the architecture and how to utilize the C++ Bridge to implement communication channels.

13.2.1Broker Configuration

This section outlines some of the steps involved in configuring the broker.

1. Download and unzip the NaradaBrokering to some local directory (say NB_HOME)
2. Start the Broker using the startbr.sh shell scripts in the bin directory inside NB_HOME.

> Note: If you need to handle larger payloads, please change the line in the startbr.sh
>
> ```
> java -classpath $cp cgl.narada.node.BrokerNode $brokerConfigFile
> $serviceConfigFile $brokerCommunicatorPort&
> ```
>
> to
>
> ```
> java -Xmx<max value>m -Xms<min value>m -classpath $cp
> cgl.narada.node.BrokerNode $brokerConfigFile $serviceConfigFile
> $brokerCommunicatorPort&
> ```
>
> A benchmark test, where the broker is fired with 64kB of data at a rate of ~4.5MB, shows that <max value> of 512 is a good heap size.

13.2.2Compiling the C++ Bridge

Please note that the: Java classes for the bridge are pre-compiled and are in the **nbcppbridge.jar** located in **BRIDGE_HOME/lib**

1. Download and unzip the cppbridge.tar.gz to a local directory (say **BRIDGE_HOME**)

2. Inside **BRIDGE_HOME** you will find a **src** directory which contains both Java and the C++ code.
3. Set the **JAVA_HOME** variable in the make file (located in the **BRIDGE_HOME/src** directory) to point to the appropriate location.
4. Default goal will perform the necessary compilation, build chat executable and move it to **BRIDGE_HOME/build** directory.

13.2.3 Simple Chat Client

1. Once the C++ Bridge is compiled go to **BRIDGE_HOME/build** directory and run the chat.sh to start a chat client. e.g. **./chat.sh 5000**

2. The integer argument is the entity Id which identifies this client in a given broker network. To start the second chat client run it again in a new shell with different entity id. (say **./chat.sh 6000**)

3. Once the two chat windows shows the line "Happy Chatting", you can type any text to be sent to the other.

4. To exit from the chat client type **$<return>.**

13.2.4 The Architecture

The C++ Bridge uses JNI technology to communicate with NaradaBrokering. The following diagram shows the high-level architecture.

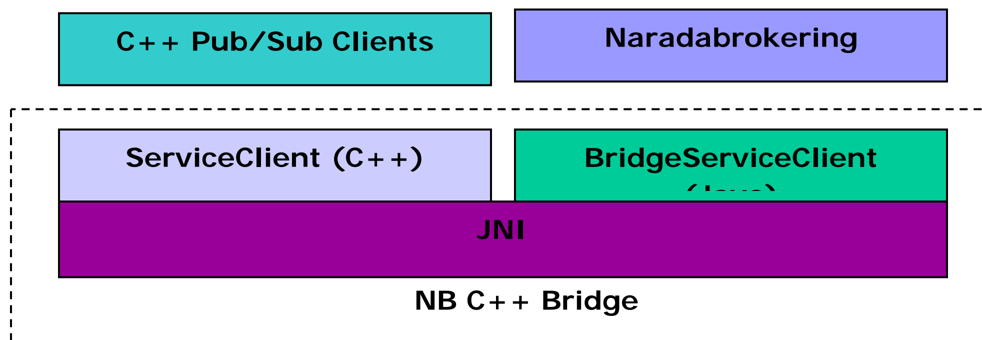


Figure 2: Architecture of the C++ Bridge for NaradaBrokering.

13.2.5 How to Use the Bridge

The API that the C++ programmer needs to work with comprises one class - **ServiceClient** and the **Callback** interface.

The **ServiceClient** contains the following public methods and perform the tasks as described below.

```
bool init (int entity_id, char *config_file_path, char
*host_name,  int port_num, char *transport);
```

Description:
This will initialize the **ServiceClient** where it will load the JVM and initializes communications with the broker.

Parameters:
`entity_id` – ID of this service instance
`config_file_path` - Path to the ServiceConfiguration.txt
`host_name` - Host Name where the Broker is running
`port_num` - Port Number of the Broker
`transport` - Transport type to be used. (Default uses TCP) possible options "niotcp" , "udp"

```
bool subscribe (char *topic, long callbackId);
```

Description:
This is used to subscribe to any topic using this **ServiceClient** instance.

Parameters:
`Topic` – Topic to which the messages are sent. **e.g. "/topics/nbcpp"**
`callbackId` - This is the reference to the Callback object used for this topic.

User can provide different callback objects for different topics or use the same callback object. The `callbackId` should be a pointer to any implementation of **Callback** interface. Please see the **chat_client.cc** for a sample.

```
bool publish (char *topic, char *transfer_bytes);
```

Description:
This will publish a given set of bytes to a given topic using this **ServiceClient** instance.
`topic` - Topic to which the message is published. e.g. **"/topic/nbcpp"**

`transfer_bytes` - Set of bytes to be transferred.

> Note: If multiple publishers and subscribers need to be run in a single process they should all share a single instance of **`ServiceClient`** as it is not possible to create multiple JVMs in a single process.

13.2.6Simple Publisher Example

The following code fragment shows the methods that need to be used in order to write a publisher using the above API.

```
int
main (int argc, char *argv[])
{
  //Input parameters for publisher
  int entity_id = atoi (argv[1]);
  char *service_config_path
=/test/abc/ServiceConfiguration.txt";
  char *host_name = "gf6.ucs.indiana.edu";
  int port_num = 3075;
  char *transport = "niotcp";
  char *topic = "/publish/mytopic";
  ServiceClient sClient;


 //Initialize the service_client
  if (!sClient.
      init (entity_id, service_config_path, host_name, port_num,
    transport))
    {
      cout << "Error:Initialization Failed \n";
    }

   //Publish a given set of bytes.
   char *buffer="This is my test message";
   if (!sClient.publish (topic, buffer))
   {
     cout << "Error:Publishing Failed \n";
   }

  return 0;
}
```

14 Appendix A: Working with the codebase in IDEs

14.1 Incorporating the NaradaBrokering Codebase into Eclipse

In this section of the user's guide we describe how to import the NaradaBrokering codebase into the Eclipse IDE. The version of eclipse platform that we use for our descriptions is 3.2. There may be minor differences to the steps if your version of eclipse is different. However, we expect that the overall process will be similar.

14.1.1 Download NaradaBrokering and the Necessary Jars

Download NaradaBrokering from www.naradabrokering.org and extract the zip file to a separate directory. The NaradaBrokering zip file contains all the necessary jar files (except **jms.jar** and **jmf.jar**) in its **lib** directory. Follow instructions in section 1 to retrieve these files.

14.1.2 Creating New Project Using Eclipse

Start the Eclipse programand select **File-> New-> Project** as shown below [Figure 41].

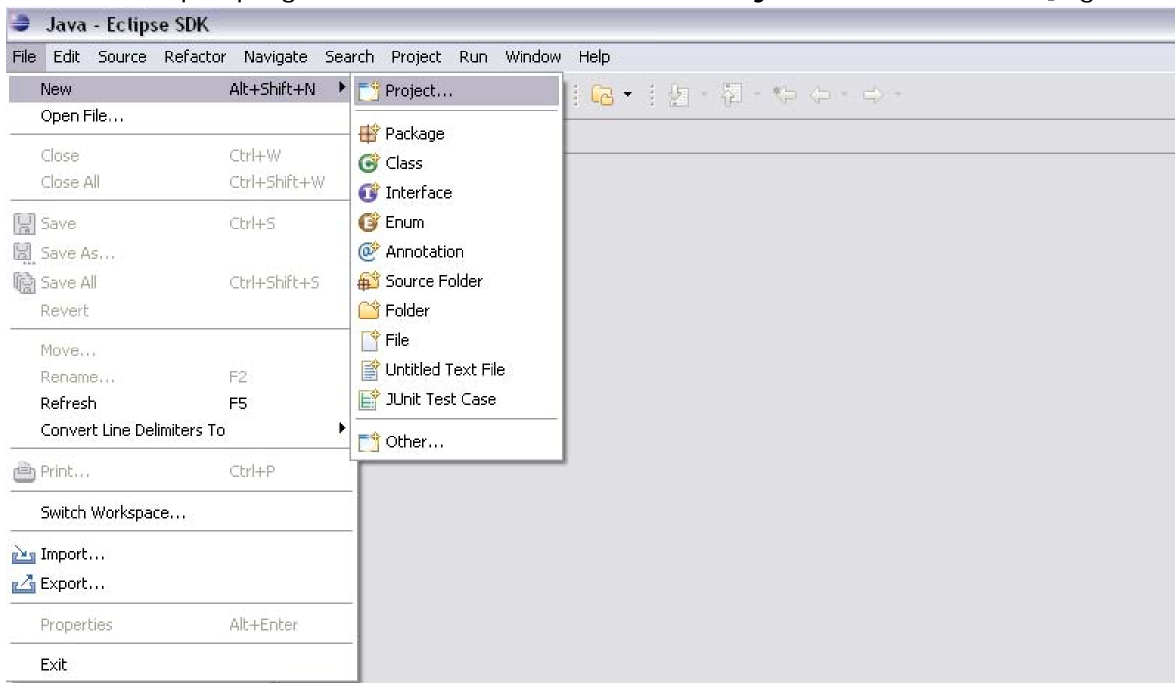


Figure 41: Creating new project using eclipse.

The next step is to select the type of the project you need. Select **Java Project** and press **Next** as shown in the following screenshot [Figure 42].

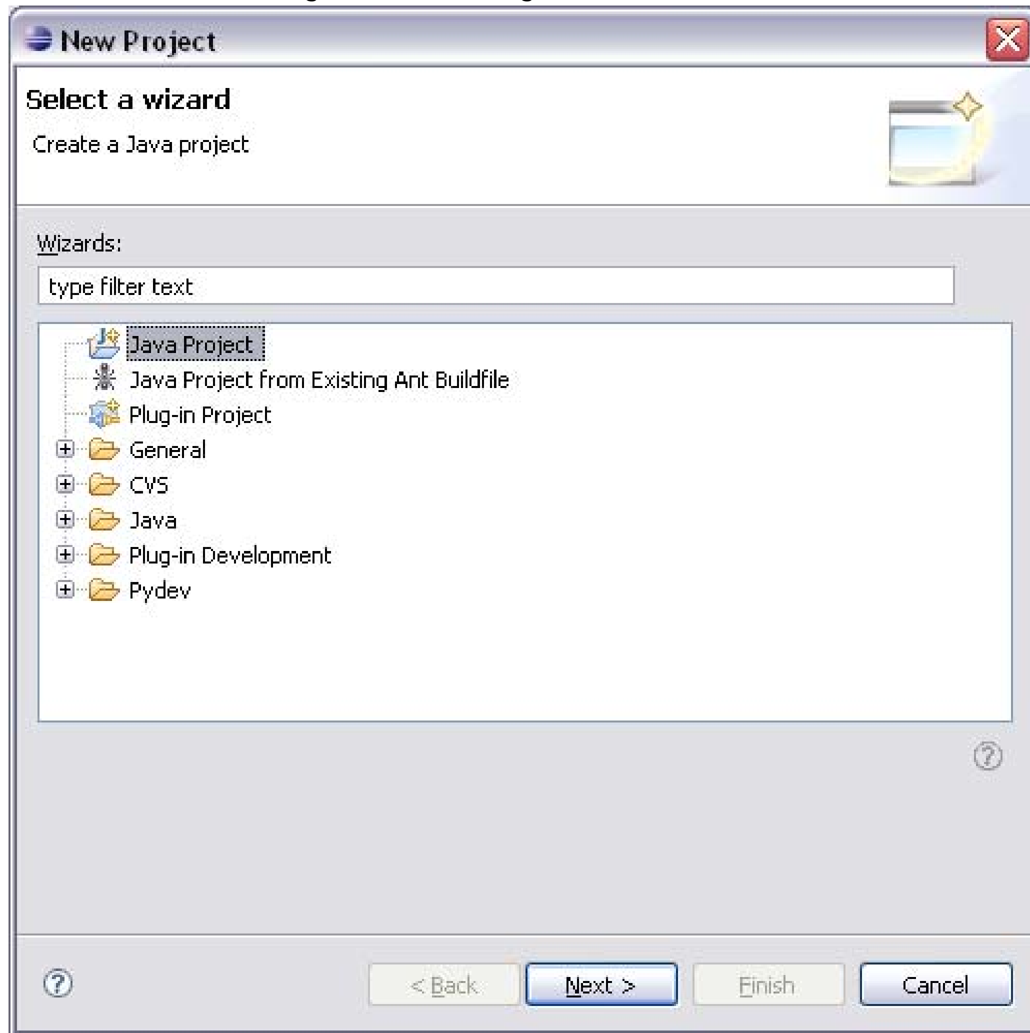


Figure 42: Selecting Java Project as the project type.

In the next window [Figure 43] you can specify the name of the project and the location of the project to be created. For the project location, please select the option "***Create project from existing source***". Since you have already unzipped the NaradaBrokering source code into a directory, you can select that directory using the **Browse..** button.

Figure 43: Specifying project name and location.

Once you specify the name and the location of the project please press **Next** and you will be prompted with the following screen [Figure 44].

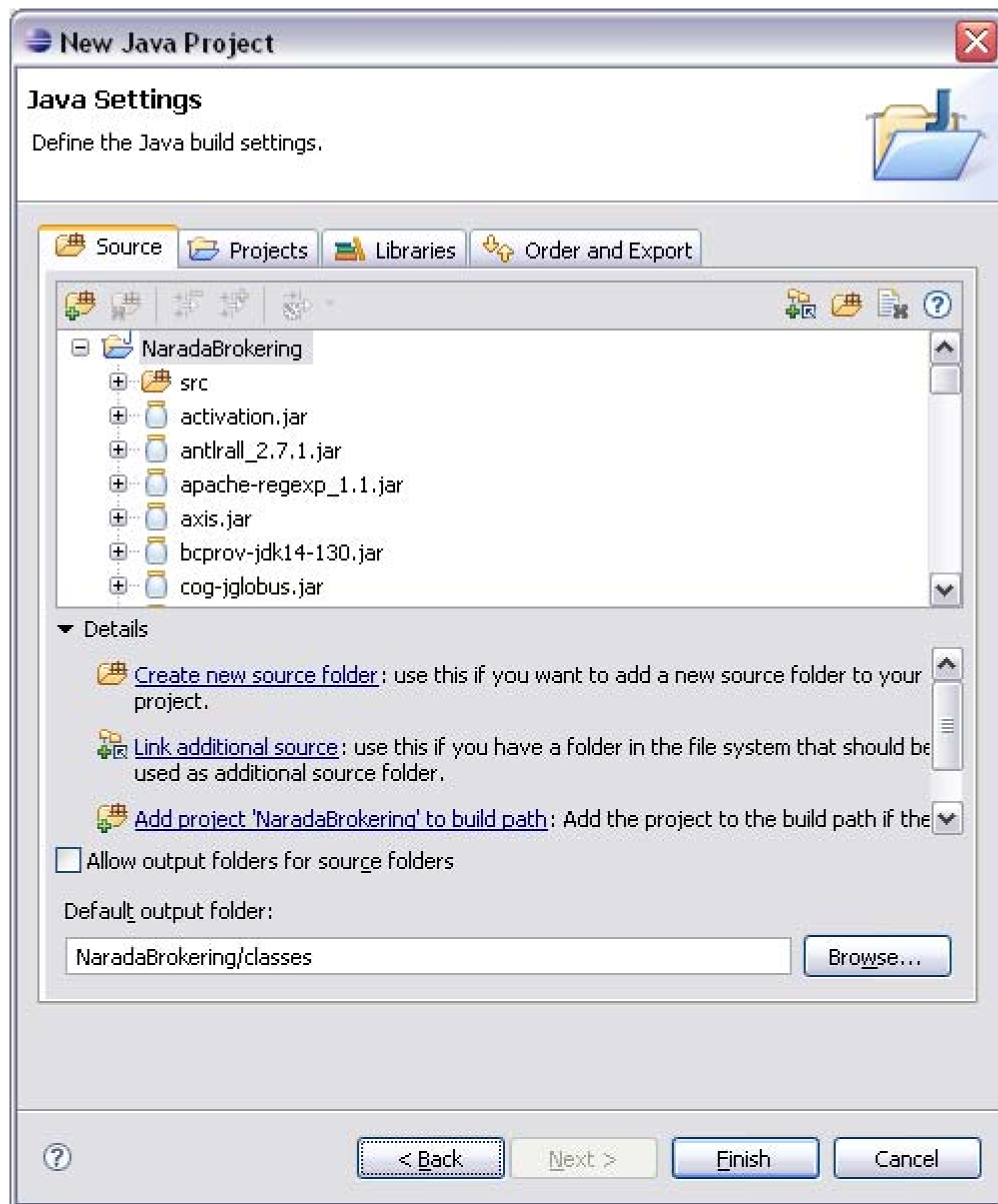


Figure 44: Source directory and output directory of the project.

By this time you will see that the Eclipse IDE has already identified the source directory, the libraries and the output directory for the project. Before finishing the project creation, there is one more step you should perform.

Select the **Libraries** tab and remove the NaradaBrokering.jar from the project. (Since you have already mounted the NaradaBrokering codebase the classes that you compile will be newer than the classes available in this jar file.) This step is shown in the following diagram [Figure 45].

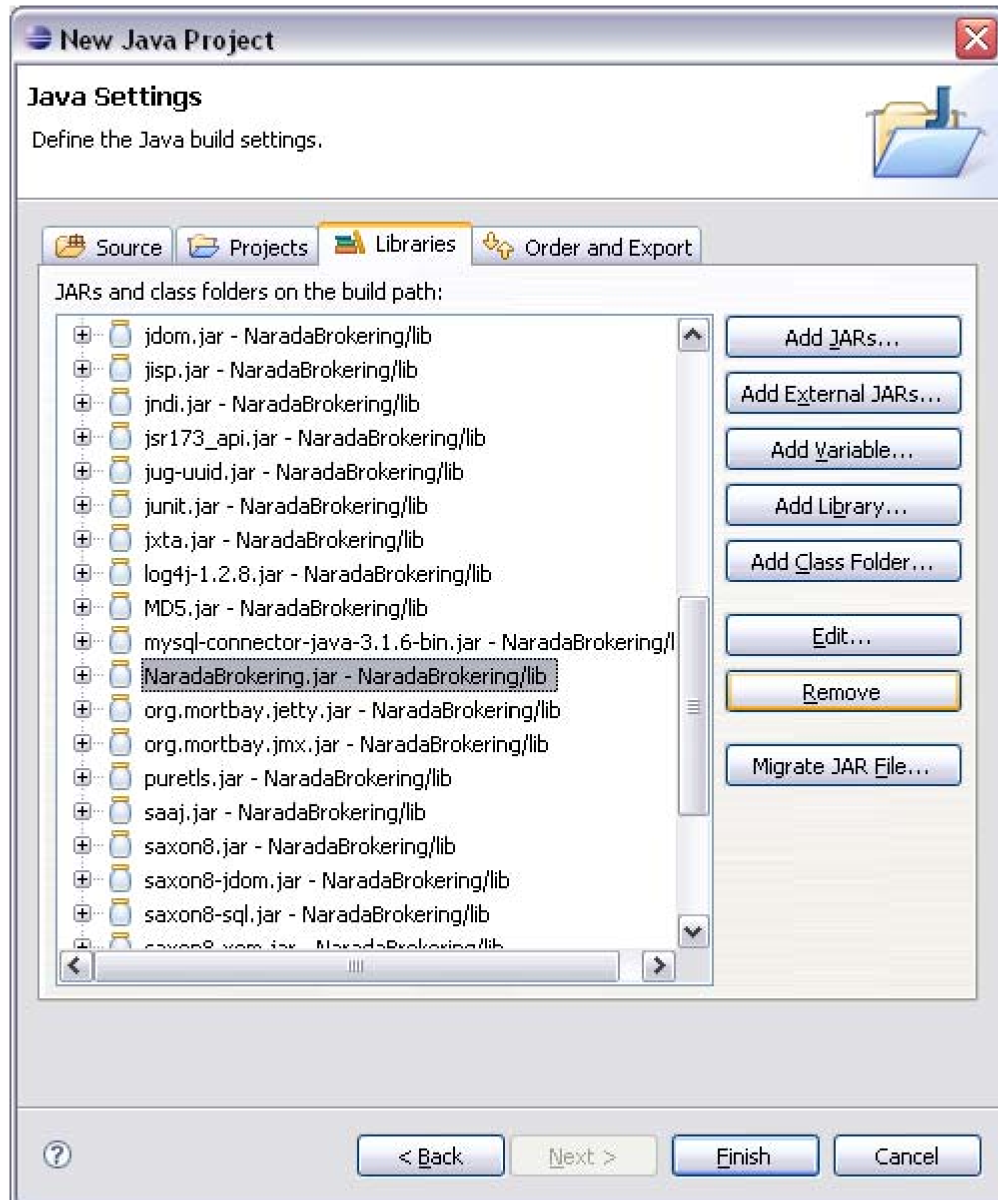


Figure 45: Removing NaradaBrokering.jar from the project references.

After this please press **Finish** button to end the project creation step. Once this is done, you will see that NaradaBrokering codebase is correctly imported into the Eclipse IDE as shown below [Figure 46].

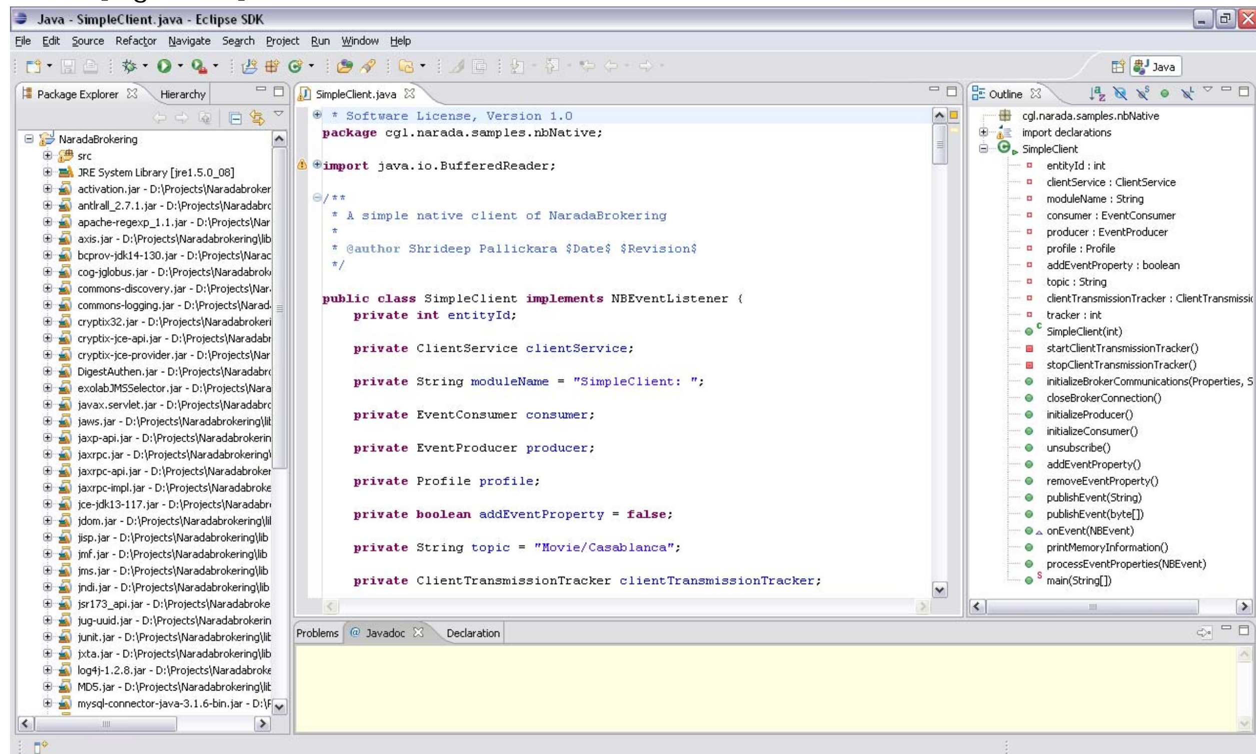


Figure 46: After importing NaradaBrokering to eclipse.

14.1.3Use NaradaBrokering in Your Project

This section of the user's guide will describe how to use features of NaradaBrokering in your project with the Eclipse IDE. Here we assume that you already have an eclipse project to which you need to add features of Naradabrokering. We also assumed that you have downloaded the latest version of NaradaBrokering and unzipped the content to some directory in your machine.

The only step required to use NaradaBrokering in your project is to add the necessary jar files to the project. Please select **Project->Properties** from the main menu of Eclipse as shown below [Figure 47].

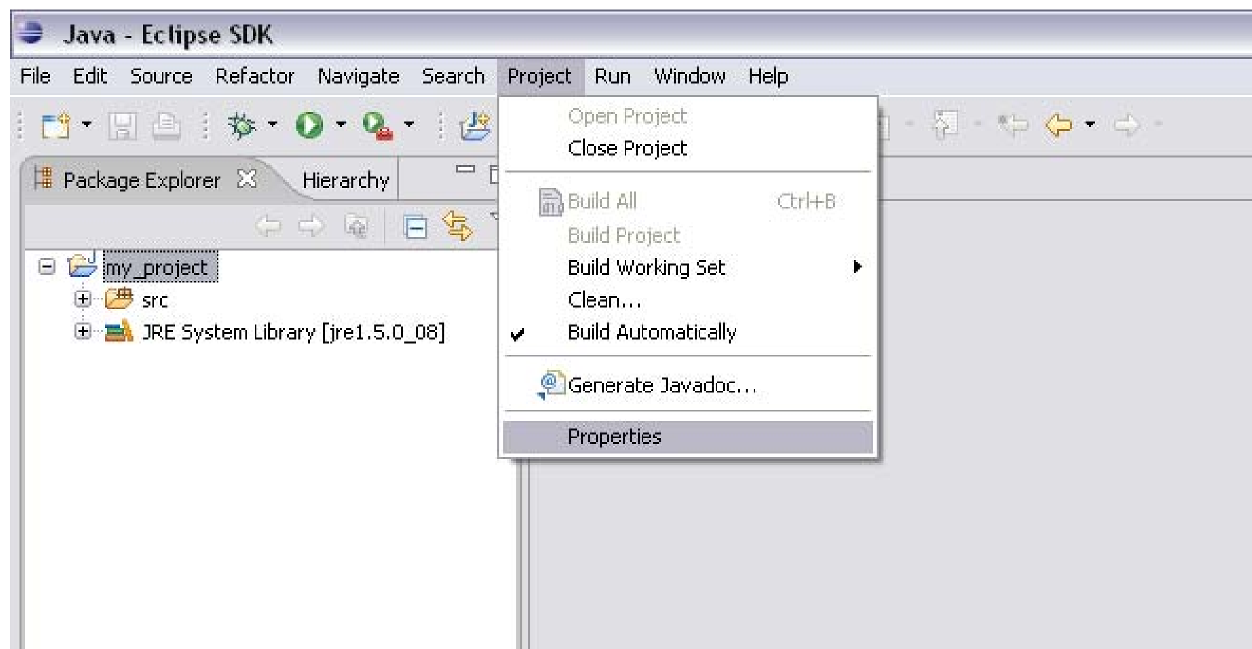


Figure 47: Selecting project->properties.

Once you click the properties menu option, you will see the following window [Figure 48]. Please select the **Libraries** tab of that window.

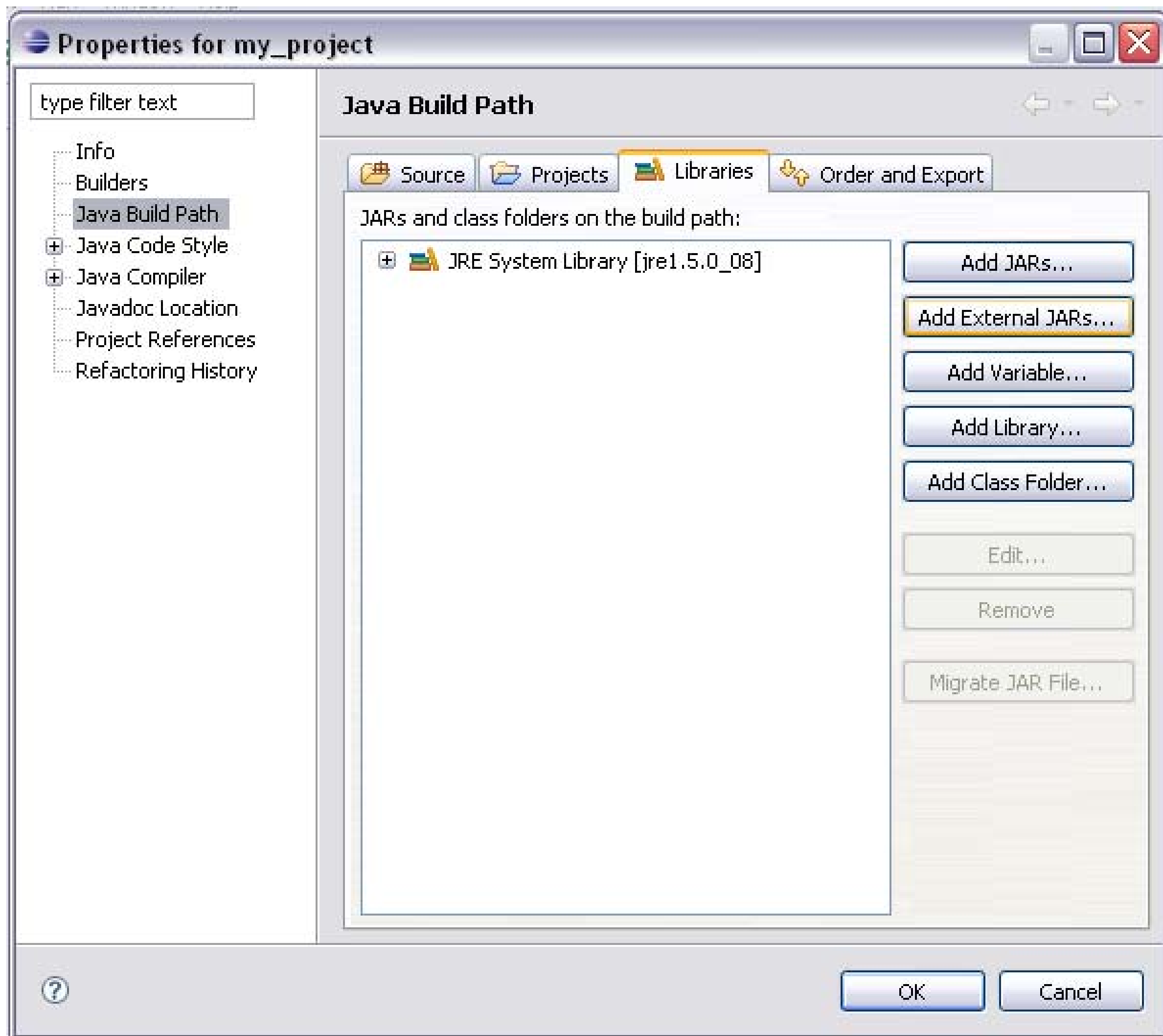


Figure 48: Adding jar files using project configuration panel.

You can use this window to add NaradaBrokering specific jar files to your project. Please select **Add External Jars..** button to browse and select jar files. Locate the **lib** directory of NaradaBrokering distribution. Select all the files in this directory as shown below [Figure 49].

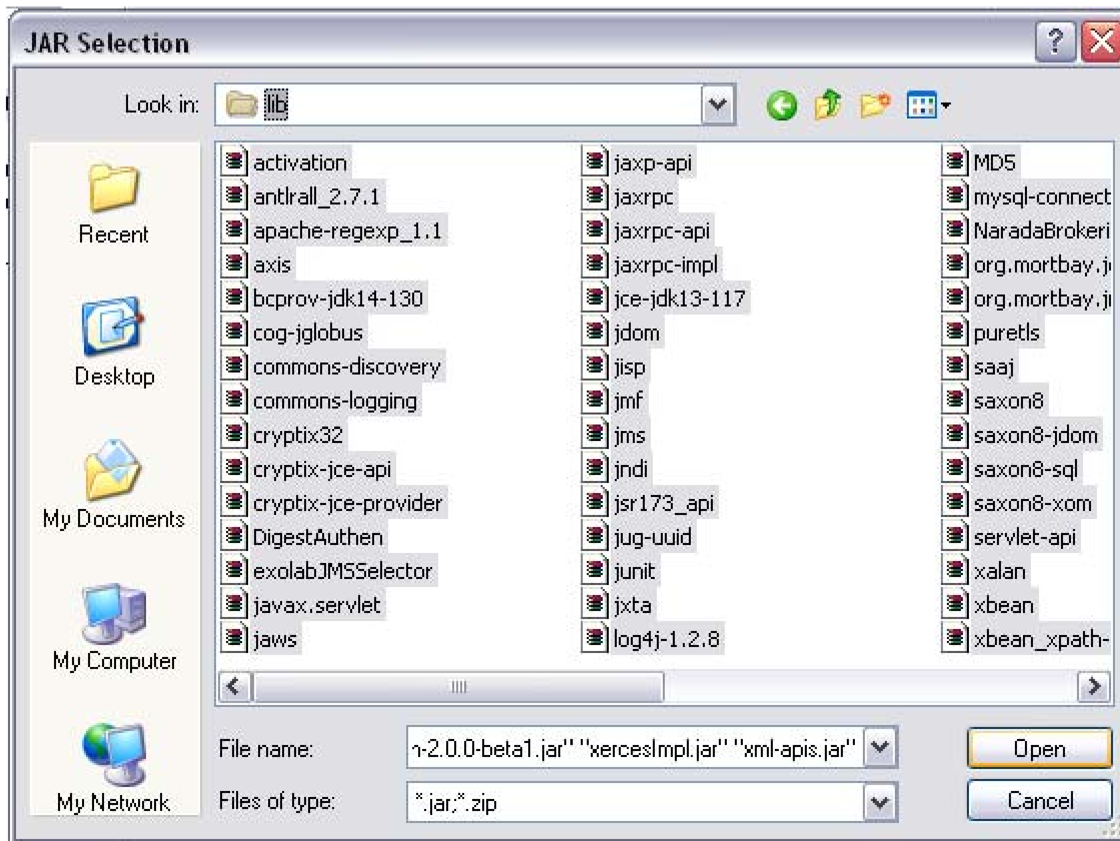


Figure 49: Selecting all the jar files in "lib" directory.

Press **Open** to add all the jar files to the project. Now you will see that your project contains references to those jars that you have selected [**Error! Reference source not found.**].

14.2 Importing the codebase into JBuilder

First update the JDK used in JBuilder to point to the appropriate JDK (see the requirements section 1.1.1). To do this, go to Tools| Configure JDKs. Then select New. You can now specify your JAVA_HOME.

Next start a new project using File|New Project and also specifying the directory in the Project Wizard to point to the %NB_HOME% variable. Next, in your project paths you have to make sure that you select the new JDK. Exit the wizard.

Create a **src** directory in your project home directory and move the **cgl** directory in the NaradaBrokering's distribution to the **src** directory. Now, if you select Project|Refresh, you can see all the packages in the left pane of JBuilder.

Create NaradaBrokering libraries for your JBuilder project. Select **Tool|Configure Library|New**. Click **Add**. Select all the ".jar" files from the **lib** directory in NaradaBrokering distribution and click OK. Then give a name to the new library, e.g., NaradaBrLib. Then click OK to finish.

Add Narada Library to you project. Select **Project|Project Properties**. Then select the **Required Libraries** tab. Then click "**Add**". Select the **NaradaBrLib** you just created and click "OK", the library will be added to your project. Now, select **Project|Rebuild Project**. You should compile the project successfully.

To run the test program, you need to add them to your run configuration and also add the **Application Parameters**.

15 Appendix B: The Broker Configuration File

```
#This is the Non Blocking TCP port to which the broker listens for
connections.
NIOTCPBrokerPort=3045

#This is the TCP port to which the broker listens for connections.
TCPBrokerPort=5045

#This is the UDP port to which the broker listens for connections. It is
# a good idea to have this port number be #identical to the TCP port.
#The UDP communication is used specifically for transient events, since
#there are no error corrections for UDP based communication.
UDPBrokerPort=3045

MulticastGroupHost=224.224.224.224
MulticastGroupPort=4045

#This is the Non Blocking Thread pool TCP port to which the broker
listens for
#connections.
PoolTCPBrokerPort=6045

#This specifies the limit on concurrent connections. Base it on the
#capabilities of the machine hosting the broker.This is also used by the
#broker locator to determine the best available broker.
ConcurrentConnectionLimit=3000


#If this is a stand alone node, this should be "true". If this broker
#node is intended to be the first node within a #distributed setting
#this should be "true". If this node is to receive its address
#from another broker, this should be "false".
AssignedAddress=true

# This gives the Geographical / Institutional info about this broker
AboutThisBroker=CGL, Indiana University, Bloomington, IN, U.S.A.

# Comma seperated list of publicly known BDNs (listed in preference
Order)
# BDNList=http://www.idonotexist.com,
#http://trex.ucs.indiana.edu:8080/BDN/servlet/BDN,
#http://www.gridserlocator.org/
# BDNList=http://trex.ucs.indiana.edu:8080/BDN/servlet/BDN
```

```
BDNList=


# Broker Discovery Request Response Policy
DiscoveryResponsePolicy=cgl.narada.discovery.broker.
DefaultBrokerDiscoveryRequestResponsePolicy

# A String (or UUID) referring to the private broker network ID to which
this broker belongs
# This value if missing OR * => this is a public broker
VirtualBrokerNetwork=network-CGL-1
# VirtualBrokerNetwork=*

# Maximum number of requests to store
MAXBrokerDiscoRequests=1000
```