Shared Data Toolkit for Java Technology User Guide

This is a toolkit defined to support highly interactive, collaborative applications written in the Java programming language.

Version 2.3   November 14, 2017

Please send technical comments on this user guide to:

jsdt.interest@gmail.com

Rich Burridge, JSDT Author

Table of Contents

1 Introduction

1.1 What Is Collaborative Computing?

Collaborative computing means different things to different people. Some of the areas covered are:

- Application Sharing
  - Takes an existing single-user application and makes it shareable
  - Broadcasting graphics, mouse movements, and edits to all participants
  - Input focus control sharing, floor control
  - Telepointers and “Master” pointer
  - Integrated with audio, video, text chat connections (session management)
- Multi-User Application Toolkits
  - Enabling developers to create multi-user applications
  - Shared widgets
  - Multiple views of shared space (radar views, WYSIWID, miniature)
  - Multi-user scrollbars, presenting each user’s viewing area
- Interactive Desktop Conferencing
  - Setting up sessions, audio, video, shared application connections
  - Adding late joiners, more than 2-way connections
  - Migrating to other ways of interaction (asynchronous, subgroups)
  - Integration of other media (phone conference, PictureTel)
- Distributed Presentations over the Net
  - This involves broadcasting a presentation via audio, video, and electronic slides to participants at their computer desktops, and allowing the audience to interact with the speaker and other audience members
  - Multicasting audio, video, graphics, shared text
  - Enabling audio, video, shared text, graphic feeback
  - Enabling side conversations among audience members
  - Shared widgets (poll meters)

    - Large scale (100’s to 1000’s) and asymmetric (mostly 1 to many)
    - Recording and playing back events is also of interest
- People Objects
    - A standardized way of representing contact information for people, “electronic business card”
    - Making it easy to find people, establish contact with them, coordinate with their schedules
    - Easily and flexibly identifying and forming groups (e.g., creating mail distribution lists by collecting people objects)
    - Attaching people objects to all electronic contributions, so it is easy to contact the person that is behind the electronic information
    - Beyond static information (address, phone number), also represent dynamic information (what kind of activity users are doing, if they’re out of the office)
- Awareness
    - Being aware of other people that you work with in a way that enables impromptu, unintended encounters with them
    - Need information about where users are, what they’re doing
    - Knowing activity status of others’ machines, devices, peripherals
    - Upon becoming aware of someone you want to contact, it must be easy to migrate to interactive conferencing or communication
- Virtual Worlds
    - Creating a virtual place populated with avatars that can navigate and interact with other people and objects in the environment
    - Persistent places
    - Containment and tracking of objects
    - User extensible
    - Shared video and audio, spatialized audio, selective groupings of users
    - Multi-user text chat, MOO
- Workflow
    - Coordinating asynchronous transfer and development of information
    - Version control
    - User permissions

    - Synchronization
    - Notification of new material
    - Group calendaring
- Social Filtering
    - Beyond just providing recommendations from a large database, trying to match people’s interests to provide tailored recommendations from others with similar interests
    - This probably involves specifying API’s for applications to consult with other applications and other people objects
    - Probably also involves API’s for agents
- Computer-Augmented Meeting Rooms/Group Decision Support Systems (GDSS)
    - Enables groups to meet in a room outfitted with a large shared whiteboard device networked with individual workstations and personal devices
    - Device integration (Liveboard, workstation, PDA)
    - Integrating multiple voting, organizing tools

1.2 Abstract

This toolkit has been defined to support highly interactive, collaborative applications. It provides the basic abstraction of a session (i.e., a group of objects associated with some common communications pattern), and supports full-duplex multipoint communication among an arbitrary number of connected application entities -- all over a variety of different types of networks. In addition, this tookit provides efficient support of multicast message communications, the ability to ensure uniformly sequenced message delivery and a token-based distributed synchronization mechanism. It also provides the ability to share byte arrays amongst the members of a session.

1.3 Overview

The primary functionality provided by the communications mechanism described here is the ability for collaboration-aware code in the Java™ programming language, to send data to all (or a subset) of the participants within a communications session. This is accomplished by way of various send methods. The ability to share byte arrays amongst the various member of the session is also provided. In addition, a token mechanism is included which provides the basis for the construction of a wide range of application-level synchronization facilities e.g., the ability to ensure mutually exclusive access to a resource, to perform distributed, multi-application, atomic signaling, etc.

It is intended that this toolkit provide a common interface for general multi-party communications, beneath which a wide variety of implementation technologies can be employed. In particular, the specific protocol stack used to implement the functionality defined by this toolkit, as well as the negotiation process used to select a specific protocol, are not visible to the user of this interface. Therefore, a range of different protocols can be hidden within the implementation of this interface (including standards-based multi-party communications protocols (e.g., T.12x), custom protocols based on standard networking interfaces (e.g., TCP/IP), and arbitrary proprietary protocols).

The definition of this interface involved an explicit effort to make use of as many existing concepts from the Java technology as possible -- most notably, perhaps, the event model from the JDK™ 1.1 release. Also, where possible, this interface has borrowed text, ideas, and definitions from the ITU T.122 recommendation for Multipoint Communication Service for Audiographics and Audiovisual Conferencing Services Definition.

2 Getting Started

2.1 Overview

There are various components that make up a JSDT application. This section describes them all, introducing them in the order you are likely to encounter them. For more information on each JSDT class and the methods it offers, please also review the associated JavaDoc documentation. The JSDT distribution also comes with several complete examples which include source code. These are a good alternative source of information to see how to put all this together.

2.2 URLString

As with all types of collaborative environments, there needs to be some way for each application to initially rendezvous, so that data can be shared. The JSDT group rendezvous point is a Session object, and a special kind of URL string of the form:

jsdt://<*server*>:<*port*>/<*impl*>/Session/<*name*>

is used, where:

<*server*> is the name of the server computer for this JSDT Session.
<*port*> is the port number to use for connections.
<*impl*> is the implementation type.
<*name*> is the name of the Session.

Alternatively, an IP address can be supplied as the name of the server computer.

A URLString class is provided which is used to create these special URL strings from their component parts. It can also be used to extract out the individual parts of a URLString.

There is typically two parts to a JSDT application: the server and one or more proxies. The server is a process that will receive messages from the proxies, process them, and send out further messages to the other proxies or a reply to the proxy that sent the initial message. The server is always started before any of the proxies, and just after the Registry (see Registry).

JSDT currently comes with two implementation types:

• socket	- uses TCP/IP sockets
• http	- uses HTTP commands

Each implementation type uses a different transport protocol.

The URLString class contains a static convenience method for creating these Session URL Strings. It is:

```
public static URLString
createSessionURL(String hostName, int port,
                 String connectionType, String sessionName);
```

For example, a URLString for a chat Session could be created with:

```
URLString urlString =
    URLString.createSessionURL(“stard.Eng.Sun.COM”, 3355,
                               “socket”, “chatSession”);
```

Internally, this would be equivalent to:

```
“jsdt://stard.Eng.Sun.COM:3355/socket/Session/chatSession”
```

Note that the URL String is encapsulated in a URLString object, not a java.net.URL object.

Multiple Sessions can use the same *<server>*:*<port>* pair. Communication between the server and its proxies for each of those Sessions will be multiplexed over the same connection.

It is also possible to contact a Client that has registered itself with the Registry (see Inviting and Expelling Clients”).

2.3 Registry

The information for each Session needs to be kept somewhere that is easily accessable to applications. This is where the Registry fits in. The Registry can be started either in its own Java runtime environment, or as a thread within the server, on the host that is the server for each JSDT Session or Client. The Registry contains a transient database that maps names to JSDT objects. There are two types of JSDT object that can be stored in the Registry; a Session and a Client. When the Registry is first started its database is empty. The names stored in the Registry are pure and are not parsed. A collaborative service storing itself in the Registry may want to prefix the name of the service by a package name (although this is not required), to reduce name collisions in the Registry.

By default, the JSDT Registry uses port 4561 to communicate with JSDT applications.

Starting the Registry can be done separately, in other words:

`system%` **java com.sun.media.jsdt.*$(TYPE)*.Registry**

where ***$(TYPE)*** is the implementation type you are using for your collaboration (one of “socket”, “http” or “multicast”).

For example:

system% **java com.sun.media.jsdt.socket.Registry**

Or you can use the RegistryFactory class to start the Registry.

2.3.1 RegistryFactory

If you wish to start a Registry of the appropriate type from within your JSDT server application, running as a separate thread, then you can use the RegistryFactory class. The RegistryFactory can be used to:

- start a Registry
- stop a Registry
- check if the Registry already exists
- add or remove a Registry listener
- list the contents of the Registry

If a port number is not specified, then the current value of the `com.sun.media.jsdt.impl.JSDTObject.registryPort` variable will be used.

You need to know the implementation type you will be using. You can determine if a Registry of the appropriate implementation type is running using one of the registryExists methods. If it’s not running you can start it with one of the startRegistry methods. For example:

```
import com.sun.media.jsdt.*;

String type = “socket”;

try {
     if (RegistryFactory.registryExists(type) == false) {
          RegistryFactory.startRegistry(type);
     }
} catch (NoRegistryException nre) {
           System.out.println(“Couldn’t start a Registry of this
type.”);
      } catch (RegistryExistException ree) {
     System.out.println(“The Registry is already running.”);
}
```

The first thing to note here is the import line in the example above. This imports all the class files for the com.sun.media.jsdt package. You will need this whenever you are writing JSDT code. This line will be omitted in the rest of the coding examples.

Note also that various JSDT methods throw different exceptions. We need to catch these exceptions, and handle them appropriately. More on exceptions in a later chapter.

If the Registry is started with the RegistryFactory.startRegistry method, it can have a manager associated with it. This manager is used for authentication purposes every time a Client tries to create or destroy a Session or Client.

There is one big catch with starting a Registry using one of the startRegistry methods. If the application that called startRegistry is terminated, the Registry process is terminated also. This isn’t a problem if there is only one type of JSDT application using Sessions or Clients within the Registry, but if others were using it, they will no longer work. The workaround in this case is to start the Registry as a separate process.

2.4 Client

A Client is an object which is part of a JSDT application or applet and is a participant in an instance of multiparty communications. Once properly associated with one another (see Session”), related Clients can transfer data in a point-to-point or multipoint fashion.

A Client object can be the source or the destination of the data which is being exchanged in an instance of communication. It is also used when any kind of authentication is needed.

Any number of objects in an applet or application in the Java programming language, can be defined to be Client objects (with respect to this multiparty communications toolkit).

The Client interface needs to be implemented by any object that is going to be a participant in a Session. A Client declares two methods:

```
public Object authenticate(AuthenticationInfo info);

public String getName();
```

The authenticate method is used for authentication purposed by the Manager of any managed objects (see Chapter 3, “Managers”). If you are not using managed objects, then you can just return null here. The getName method needs to return a String which will be the name of this Client. It will need to be unique within any object the Client joins. It cannot be null. So a minimal implementation of a class which implements Client would look something like:

```
public class
ExampleClient implements Client {

    private String name;

    public
    ExampleClient(String name) {
        this.name = name;
    }

    public Object
    authenticate(AuthenticationInfo info) {
        return(null);
    }

    public String
    getName() {
       return(name);
    }
}
```

Instantiating such an object would be something like:

```
Client client = new ExampleClient(“Jane”);
```

2.5 *ClientFactory*

JSDT provides a factory class for creating special Clients. These don't just implement the Client interface; they are also capable of receiving messages sent to them, and processing them appropriately. They are used primarily to invite or expel a Client from a Manageable object (see Inviting and Expelling Clients), or to give a Token to another Client (see Section 6.1, "Giving a Token).

The ClientFactory can be used to:

- create a special Client
- get a reference to an existing Client
- destroy a Client
- check if a Client exists
- lists all the special Clients created and bound in the Registry

As well as being able to store references to Sessions in the Registry, Client references can also be placed there. These references can be looked up to get a handle to that Client.
The format of the URL String for a Client is:

```
jsdt://<server>:<port>/<impl>/Client/<name>
```

where:

<server> is the name of the computer that will act as the server for this JSDT Client. In other words, the name of the machine that is running the Registry where this special JSDT Client will be bound.

<port> is the port number the Client is running on. This is a port on the local machine; not the port number of the Registry.

<impl> is the implementation type.

<name> is the name of the Client. The name used here must be the same as that return by doing a Client.getName() method call, otherwise an InvalidClientException is thrown.

The URLString class contains a static convenience method for creating these Client URL Strings. It is:

```
public static URLString
createClientURL(String hostName, int port,
                String connectionType, String clientName);
```

So, for example, a URLString for a special Client called "fredClient" could be created with:

```
URLString urlString =
    URLString.createClientURL(“capra.Eng.Sun.COM”, 4477,
                              “socket”, “fredClient”);
```

Internally, this would be equivalent to:

```
“jsdt://capra.Eng.Sun.COM:4477/socket/Client/fredClient”
```

You can place one of these special Clients in the Registry by using the ClientFactory.createClient method.

You can lookup one of these Clients (get a reference to it), assuming it has already been created, by using the lookupClient method.

If you need to remove a Client entry from the Registry, you can use the destroyClient method.
Use the clientExists method to check if a Client already exists in the Registry.

The listClients methods are used to list all of the Clients currently in the Registry (optionally specifying a particular host to check the Registry contents on).

If you are trying to create or destroy a special Client in a managed Registry, then the Client attempting the creation will be authenticated to determine if it is permitted to perform this operation.

See Inviting and Expelling Clients, for a example using a special Client.

2.6 Session

A Session is a collection of related Clients which can exchange data via defined communications paths (see Channel” and Data”). The Session maintains the state associated with the collection of clients and their associated communications paths, and may interact with an object which encapsulates a defined session management policy (see Manager below). An application or applet can have multiple Client objects associated with the same (or different) Session objects. X
Within a Session, Clients can use ByteArrays, Channels and Tokens to share data.

A Session can be used to:

- create ByteArrays, Channels or Tokens
- determine if a ByteArray, Channel or Token exists
- determine if a ByteArray, Channel or Token is managed
- determine which ByteArrays, Channels or Tokens a Client has joined

- list it’s ByteArrays, Channels or Tokens
- add or remove a Session Listener (see Section 4.2, “Session Listener”)
- close the Session

A Session is also a Manageable object, the same as ByteArrays, Channels and Tokens (see Manageable Objects”). There are several operations that are common to these four types of objects which are described here.

Before a JSDT application can use any of these Session methods, it must first obtain a reference to that Session. This is where the SessionFactory comes in.

2.6.1 SessionFactory

A SessionFactory can be used to:

- create and join a Session
- destroy a Session
- check if a Session exists
- check if a Session is managed
- list all the Sessions created

Clients can get a reference to a Session and can automatically join it with the SessionFactory.createSession method. This is used to provide a means of returning a reference to the Session without having to worry about whether it already exists. The first call to createSession for a given URLString will cause the Session to be created in the Registry. Future calls to createSession will detect that that Session already exists, and provide a local reference to it.

If you are trying to create or destroy a Session in a managed Registry, then the Client attempting the creation will be authenticated to determine if it is permitted to perform this operation.

Because of the way this works, a JSDT server should always be the first   application to call the SessionFactory createSession method for each Session. This server application will run on the same machine as where you started your Registry. In other words, the name of the machine you specified in the *<server>* field of your JSDT Session URL. You can use the sessionExists method to determine if a Session (and by definition, the server that runs it) already exists.

Here’s an example of how to get a reference to a Session, from a proxy JSDT application, first making sure that the server has started. In this example, the client also automatically joins the Session.

```
boolean    created = false;
```

```
Client     client  = new ExampleClient(“jill”);
Session    session = null;
URLString  url     = URLString.createSessionURL(“stard”, 4461,
                                         “socket”, “wbSession”);

try {
    while (created == false) {
        if (SessionFactory.sessionExists(url) {
            session = SessionFactory.createSession(client, url, true);
            created = true;
        } else {
            try {
                Thread.sleep(5000);
            } catch (InterruptedException e) {
            }
        }
    }
} catch (JSDTException e) {
    System.out.println(“Couldn’t create the Session.”);
}
```

Note that all JSDT exceptions are derived from a common parent class JSDTException. Sometimes it’s easier to just catch that single exception type, rather than try to handle each exception type individually. Other times, you might wish to catch a specific JSDT exception, handling it in a certain way, and then catch the rest of them, and do some kind of default exception processing.

For example, the following code section will check to see if the port number used in the Session URL is already in use. If so, it will increment it, and try again until it successfully creates the Session (or fails for some other reason). A PortInUseException typically means that some other application already has a server socket on that particular port number. You can only have one server socket per port.

```
int     portNo  = 4461;
boolean created = false;
Client  client  = new ExampleClient(“jack”);
Session session = null;

try {
    while (created == false) {
        try {
            URLString url = URLString.createSessionURL(“stard”,
                                portNo, “socket”, “chatSession”);
            session = SessionFactory.createSession(client,
                                                   url, true);
            created = true;
        } catch {PortInUseException piue) {
            portNo++;
        }
    }
} catch (JSDTException e) {
    System.out.println("Couldn't create the Session.");
}
```

2.6.2 *Joining a Session*

Before a Client can share data, it needs to join the Session it's just got a reference to. As we saw in the last section, this can be done at Session creation time, or it can be achieved with the Session join method. Actually the join method is in the Manageable class, but Session is subclassed from   Manageable (see Manageable Objects").

The Session will typically have multiple Clients (either at the same site or at other sites). An application or applet can have multiple Clients in the same Session. Each Client might be handling a different kind of data (ie. audio vs video). A Client can be a member of multiple Sessions.

Here's some sample code for joining an unmanaged Session you already have a reference to:

```
Client client;
Session session;

try {
    session.join(client);
} catch (JSDTException e) {
    System.out.print("Couldn't join Session.");
}
```

2.6.3 *Closing a Session*

When you no longer want to reference a Session, you should use the Session.close method to terminate your association with it. This tidies up any resources that have been created for you, and closes the Session connection to the Server.

JSDT applets running in browsers specifically need to do this to terminate the specially created threads associated with that Session.

2.7 *Channel*

A Channel is a specific instance of a potentially multi-party communications path between two or more Clients within a given Session.

All Client objects which register an interest in receiving from a given Channel will be given Data sent on that Channel (see Channel Consumer”).

Any Client which possesses an object reference to a Channel is able to send Data on the given Channel, and a Client can have references to multiple different Channels.

A Channel can be used to:

- add or remove a Channel listener (see Section 4.3, “Channel Listener”)
- add or remove a Channel consumer (see Channel Consumer”)
- list all the Clients that are consuming this Channel
- determine if the channel is ordered and/or reliable
- allow a Client to join it in a specific mode
- determine if data is available, and receive it synchronously
- send data to all Clients, all other Clients or a single Client

Once the Session setup and Client attachment is completed, the last step to be performed before Data can be exchanged between all the members in a multi-point fashion, is to join the right combination of interaction Channels.

Channels are session-wide addresses. Every client of a session can join a Channel to receive data sent to it, and by joining an appropriate combination of Channels, and by consuming them, a Client can choose to receive Data sent to these Channels and ignore Data sent to other Channels.

Clients get a reference to a Channel and can automatically join it with the Session.createChannel method. This is used to provide a means of returning a reference to the Channel without having to worry about whether it already exists. The first call to createChannel for a particular Channel name will create a session-wide reference to it in the server. Future calls will return that reference.

You can use the Session.channelExists method to check to see if a Channel with a given name already exists.

Clients can join a Channel in one of three modes:

- Channel.READONLY
- Channel.WRITEONLY
- Channel.READWRITE

Clients joining in Channel.READONLY mode cannot send Data over the Channel. Clients joining in Channel.WRITEONLY mode cannot receive Data over the Channel. The default mode is Channel.READWRITE.

Here’s some example code to create and automatically join a Channel in Channel.READWRITE mode:

```
Session session;
Client  client;
Channel channel;

try {
    channel = session.createChannel(client, “ChatChannel”,
                                    true, true, true);
} catch (JSDTException e) {
    System.out.print(“Couldn’t create and join the Channel.”);
}
```

The five parameters to the createChannel call are:

- client - the Client that will be creating and potentially joining the Channel
- name - the name of the Channel to create
- reliable - indicates whether the channel is reliable. In other words whether data delivery is guaranteed
- ordered - indicates whether data sent over the channel arrives in the same order it was sent
- autoJoin - indicates if the Client is automatically joined to the Channel when it’s created

There is no reason why, a Client couldn’t create a Channel at one point, then join it at a later date. Here’s some sample code that takes this approach:

```
Session session;
Client  client;
Channel channel;

try {
    channel = session.createChannel(client, “ChatChannel”,
                                    true, true, false);
    ...
    channel.join(client);
} catch (JSDTException e) {
    System.out.print(“Couldn’t create and join the Channel.”);
}
```

Before a Client can send or receive data, it must join the Channel. It must also join the Session that the Channel was created in before it can join the Channel.

2.8 Channel Consumer

A Channel Consumer is a Client object which has registered its interest in receiving Data sent over a given Channel.

Any Client can add one or more Channel Consumers, and it is possible for a given Client object to be a Consumer of multiple Channels at the same time.

Data received in this way will arrive at the Consumer in an asynchronous manner. Data can also be received synchronously (see Receiving Data”).

The ChannelConsumer interface needs to be implemented by any object that wants to receive Data asynchronously sent over a Channel. A Channel Consumer declares one method:

```
public void dataReceived(Data data);
```

There is no reason why a JSDT application cannot have an object that implements more than one interface (for example Client and ChannelConsumer). This is quite common.

A minimal implementation of a class which just implements ChannelConsumer would look something like:

```
public class
ExampleConsumer implements ChannelConsumer {

    public synchronized void
    dataReceived(Data data) {
        ...
    }
}
```

Instantiating such an object, and adding it as a consumer on a Channel, for example:

```
Channel        channel;
Client         client;
ChannelConsumer consumer;

try {
    consumer = new ExampleConsumer();
    channel.addConsumer(client, consumer);
} catch (Exception e) {
    System.out.print(“Could not add Channel Consumer.”);
}
```

Note the use of the synchronized keyword in the dataReceived method declaration in the example class above. This is needed to prevent second or subsequent calls to dataReceived overwriting the Data for the first call. You need to make sure that you've fully finished processing each Data object before you process the next one.

2.9 Data

Data is a discrete unit of data (array of bytes) that is sent by a Client over a Channel to all of the Clients which have currently registered an interest in receiving data on the given Channel (see Channel Consumer").

A Data object contains the following:

- an array of bytes (the data)
- a length value (the length of the array of bytes)
- a priority
- the name of the Client that sent this Data
- the Channel the Data was sent over

Note that using the serialization capabilities of Java technology, a Data object can contain any other kind of object in the Java programming language ("Java object"), and be easily marshalled into an array of bytes at the sending end, and unmarshalled into the original Java object at the receiving end. An example of this is given below.

The Data class provides constructors for creating a Data object from:

- an array of bytes
- an array of bytes and a given length
- a String object
- a Java object. This Java object must be serializable

There are four Data priority levels:

- Channel.TOP_PRIORITY
- Channel.HIGH_PRIORITY
- Channel.MEDIUM_PRIORITY
- Channel.LOW_PRIORITY

The default priority level is Channel.MEDIUM_PRIORITY.

2.9.1 Sending Data

The Channel.send method provides the “one-to-many” communication, which includes point-to-point as a particular case. The sequencing of Data sent from one sender on one Channel at one priority is maintained identically at all receivers.

The Channel class provides three send methods:

```
public void
sendToAll(Client sendingClient, Data data)
throws ConnectionException, InvalidClientException,
       NoSuchChannelException, NoSuchClientException,
       NoSuchSessionException, PermissionDeniedException,
       TimedOutException;

public void
sendToOthers(Client sendingClient, Data data)
throws ConnectionException, InvalidClientException,
       NoSuchChannelException, NoSuchClientException,
       PermissionDeniedException, TimedOutException;

public void
sendToClient(Client sendingClient,
             String receivingClientName, Data data)
throws ConnectionException, InvalidClientException,
       NoSuchChannelException, NoSuchClientException,
       NoSuchConsumerException, PermissionDeniedException,
       TimedOutException;
```

The sendToAll method is used to send Data to all Clients who are consuming this Channel. If the sender is a consumer of this Channel, then it too will receive the Data.

The sendToOthers method is used to send Data to other Clients consuming this Channel. The sender (irrespective of whether it’s a consumer of this channel) will not receive the Data.

The sendToClient method is used to send Data to a single Client consuming this Channel.

Here’s an example of sending a String as a Data object over a Channel to all consumers:

```
Channel channel;
Client  client;
String  message = “Hello World”;

try {
    Data data = new Data(message);
    channel.sendToAll(client, data);
} catch (JSDTException e) {
    System.out.println("Couldn't send Data over Channel.");
}
```

2.9.2 *Receiving Data*

Data can be received over a Channel both asynchronously and synchronously. If you have setup a Channel Consumer (see Channel Consumer") and Data has been sent over a Channel to you, then the dataReceived method of that Channel Consumer will be called when the Data is received.

The Data class provides convenience methods to get the contents of the Data object as:

- an array of bytes
- a String object
- a Java object

Using the sendToAll example from the section above, a Channel Consumer could receive this and unpack it as a String with:

```
public synchronized void
dataReceived(Data data) {
    String senderName = data.getSenderName();
    String theData    = data.getDataAsString();
    String message    = senderName + ": " + theData;

    System.out.println(message);
}
```

There is no reason why a Channel Consumer cannot consume Data on more than one Channel at a time. In some situations this may be easier. Here's a code snippet for a Channel Consumer, that is handling Data received from two Channels. Here's the setup:

```
Client          client;
Channel         channel1, channel2;
ChannelConsumer consumer;

try {
    consumer = new ExampleConsumer();
    channel1.addConsumer(client, consumer);
    channel2.addConsumer(client, consumer);
} catch (JSDTException e) {
    System.out.println("Couldn't add Channel Consumers.");
}
```

Here's the dataReceived method for this Channel Consumer:

```
public synchronized void
dataReceived(Data data) {
    Channel channel    = data.getChannel();
    byte[]  theData    = data.getDataAsBytes();

    if (channel.equals(channel1) {
              ... handle data for channel 1 ...
          } else if (channel.equals(channel2) {
        ... handle data for channel 2 ...
    }
}
```

If you wish to receive Data synchronously, use the Channel.receive method. There are two variations:

```
public Data
receive(Client client)
    throws ConnectionException, InvalidClientException,
           NoSuchClientException, PermissionDeniedException,
           TimedOutException;

public Data
receive(Client client, long timeout)
    throws ConnectionException, InvalidClientException,
           NoSuchClientException, PermissionDeniedException,
            TimedOutException;
```

If a timeout value is not given, this method blocks until there is Data available to receive. You can test if this is the case with the Channel.dataAvailable method.

If a timeout period is specified then if Data is immediately available it will return with it, else it will wait until the timeout period, has expired. If no Data is available at this time, it will return null. Note that if Data becomes available during the timeout period, this method will be woken up and that Data is immediately returned.

Here’s one way of doing the synchronous equivalent of the above example:

```
Client client;

try {
    if (channel.dataAvailable(client) == true) {
                   Data   data       = channel.receive(client);
        String senderName = data.getSenderName();
        String theData    = data.getDataAsString();
        String message    = senderName + “: ” + theData;

        System.out.println(message);
    } else {
        ... do something else ...
    }
} catch (JSDTException e) {
    System.out.println(“Couldn’t receive Data over Channel.”);
}
```

2.9.3 Sending a Java Object

You can send any Java object over a Channel as long as that object is completely Serializable. Here’s some code showing you how to do this.

Here’s the sending side:

```
Client  client;
Channel channel;

public void
sendData(Object object) {

    // Turn the Java object into a Data object.
    Data data = new Data(object);

    try {
        // Send serialized object to all Channel Consumers.
            channel.sendToAll(client, data);
    } catch (JSDTException e) {
        System.out.println(“Couldn’t send object over Channel.”);
    }
}
```

Here’s the receiving side, using the dataReceived method of a Channel Consumer:

```
Object newObject = null;

public synchronized void
dataReceived(Data data) {

    try {
        // Extract Java object contained within the Data object.
        newObject = data.getDataAsObject();
    } catch (ClassNotFoundException e) {
           System.out.println(“Couldn’t find class for new object.”);
        return;
    }

    ... work with new object ...

}
```

2.10 ByteArray

A ByteArray is an object containing data (an array of bytes) that is permanently available to Clients within a Session. This global data can be written to by a Client at anytime during the life of the Session, and that new value is available to be read by other Clients.

A ByteArray can be used to:

- add or remove a ByteArray Listener (see Section 4.4, “ByteArray Listener”)
- get or set the ByteArray value

A Client can also be notified when the value of a ByteArray has changed (see Section 4.4, “ByteArray Listener”).

Note that using the serialization capabilities of Java, the value of a ByteArray object can be easily set to any other kind of Java object.

The ByteArray class provides convenience methods for getting or setting the value as:

- an array of bytes
- a String object
- a Java object. This Java object must be serializable

Clients get a reference to a ByteArray and can automatically join it with the Session.createByteArray method. This is used to provide a means of returning a reference to the ByteArray without having to worry about whether it already exists. The first call to createByteArray for a particular ByteArray name will create a session-wide reference to it in the server. Future calls will return that reference. You can use the Session.byteArrayExists method to check to see if a ByteArray with a given name already exists.

Here’s some example code to create and automatically join a ByteArray, and get it’s current value. If this ByteArray did not already exist, then it’s initially set to a single byte array of zero value.

```
Session   session;
Client    client;
ByteArray byteArray;

try {
    byteArray = session.createByteArray(client, “StockValue”,
                                        true);
         value    = byteArray.getValue();
} catch (JSDTException e) {
    System.out.print(“Couldn’t create and join the ByteArray.”);
}
```

The three parameters to the createByteArray call are:

- client - the Client that will be creating and potentially joining the ByteArray
- name - the name of the ByteArray to create
- autoJoin - indicates if the Client is automatically joined to the ByteArray when it’s created

An important point to note here is that if the ByteArray you are getting a reference to with the createByteArray method already exists, and has a different byte array value, then the ByteArray you will be returned will have that previous value. You should use the ByteArray.getValue method to retrieve the current value of the ByteArray.

There is no reason why, a Client couldn’t create a ByteArray at one point, then join it at a later date. Here’s some sample code that takes this approach:

```
Session   session;
Client    client;
ByteArray byteArray;

try {
    byteArray = session.createByteArray(client, “StockValue”,
                                        false);
    ...
    byteArray.join(client);
} catch (JSDTException e) {
    System.out.print(“Couldn’t create and join the ByteArray.”);
}
```

Before a Client can set a ByteArray value, it must join the ByteArray. It must also join the Session that the ByteArray was created in before it can join the ByteArray.

You can use the ByteArray.setValue method to set the ByteArray to a new byte array value. This byte array can contain any kind of value. It can even contain more than one value. As long as the marshalling and unmarshalling of this data is consistent, anything can be stored in the byte array. The following sample code packs various information related to a stock value into the byte array before setting it in the ByteArray.

```
Client                client;
ByteArray             byteArray;
ByteArrayOutputStream baos = new ByteArrayOutputStream();
DataOutputStream      dos  = DataOutputStream(baos);

try {
    dos.writeUTF(symbol);
    dos.writeBoolean(isValid);
    if (isValid) {
        dos.writeUTF(time);
        dos.writeUTF(stockValue);
        dos.writeUTF(change);
        dos.writeUTF(quotes);
    }
    dos.flush();
} catch (IOException e) {
    System.out.println(“Couldn’t write stock information.”);
}

try {
    byteArray.setValue(client, baos.toByteArray());
} catch (JSDTException je) {
    System.out.println(“Couldn’t set the ByteArray value.”);
}
```

Unpacking this stock information which has been stored in the ByteArray is just the reverse of the way it was written:

```
ByteArray            byteArray;
byte[]               value;
ByteArrayInputStream bais;
DataInputStream      dis;

try {
    value = byteArray.getValue();
} catch (JSDTException je) {
     System.out.println(“Couldn’t get the ByteArray value.”);
}
bais = new ByteArrayInputStream(value, 0, value.length);
dis  = new DataInputStream(bais);

try {
   String  symbol  = dis.readUTF();
   boolean isValid = dis.readBoolean();

   if (isValid) {
        String time       = dis.readUTF();
        String stockValue = dis.readUTF();
        String change     = dis.readUTF();
        String quotes     = dis.readUTF();
    }
} catch (IOException e) {
    System.out.println(“Couldn’t read stock information.”);
}
```

2.11 Token

A Token is a synchronization object which provides a unique distributed atomic operation. Tokens can be used to implement a variety of different application-level synchronization mechanisms.

Tokens provide a means to implement exclusive access. For example, to ensure in a multipoint application using various resources, that one and only one site holds a given resource at a given time, a Token can be associated with every resource. When a site wishes to use a specific resource, it must ask for its corresponding Token, which will be granted only if no one else is holding it.
A Token can be used to:

- add or remove a Token Listener (see Section 4.5, “Token Listener”)
- grab a Token (exclusively or non-exclusively)
- list all the Clients that are holding (grabbing or inhibiting) this Token
- release a Token
- test a Token’s current status
- give a Token to another Client
- request a Token from another Client

The Token.grab method allows one client to exclusively hold a given token. The Client defines the significance of this token in the application. Other Clients may use the Token.test method to determine the status at any time and may request the token from the holder with the Token.request method. The Token holder may transfer control of a token to another specified Client with the Token.give method or return a Token to a generally available status with the Token.release method.

Doing a Token.test on a Token will show it to be in one of four states:

- Token.NOT_IN_USE - a freely available Token
- Token.GRABBED - a Token exclusively grabbed by a Client
- Token.INHIBITED - a Token non-exclusively grabbed by one or more Clients
- Token.GIVING - a Token in the process of being given to a Client by another Client

Most Token operations return a status value indicating whether the operation was a success. There are two more status values that can be return apart from the four state values listed above:

- Token.ALREADY_GRABBED - an attempt was made to grab a Token that was already being grabbed by a Client
- Token.ALREADY_INHIBITED - an attempt was made to exclusively grab a Token that was already being grabbed by one or more Clients in a non-exclusive mode

Tokens are created in a similar way to ByteArrays or Channels.

Clients get a reference to a Token and can automatically join it with the Session.createToken method. This is used to provide a means of returning a reference to the Token without having to worry about whether it already exists. The first call to createToken for a particular Token name will create a session-wide reference to it in the server. Future calls will return that reference.

You can use the Session.tokenExists method to check to see if a Token with a given name already exists.

Here’s some example code to create and automatically join a Token, and get it’s current status.

```
Session session;
Client  client;
Token   token;
int     status;
try {
    token  = session.createToken(client, “FileToken”, true);
    status = token.test();

    System.out.print(“Token status is: ”);
    switch (status) {
        case Token.NOT_IN_USE: System.out.println(“ not in use.”);
                              break;
        case Token.GRABBED:    System.out.println(“ grabbed.”);
                              break;
        case Token.INHIBITED:  System.out.println(“ inhibited.”);
                              break;
        case Token.GIVING:    System.out.println(“ giving.”);
                              break;
    }
} catch (JSDTException e) {
    System.out.print(“Couldn’t create, join or test the Token.”);
}
```

The three parameters to the createToken call are:

- client - the Client that will be creating and potentially joining the Token
- name - the name of the Token to create

- autoJoin - indicates if the Client is automatically joined to the Token when it’s created

There is no reason why, a Client couldn’t create a Token at one point, then join it at a later date. Here’s some sample code that takes this approach:

```
Session session;
Client  client;
Token   token;

try {
    token = session.createToken(client, “FileToken”, false);
    ...
    token.join(client);
} catch (JSDTException e) {
    System.out.print(“Couldn’t create and join the Token.”);
}
```

Before a Client can grab a Token, it must join the Token. It must also join the Session that the Token was created in before it can join the Token.

A single Token may be used to coordinate a multiple Client event by using the Token.grab method in a non-exclusive mode. Clients can independently inhibit and release the same Token. For example, if it was desired to know when all Clients have completed reception and processing of a bulk file transfer, all receiving Clients would non-exclusively grab (inhibit) the same Token and each individual Client would release the Token when it had completed the proscribed process. Any Client could test the Token at will to determine if the Token is free which means all the Clients have completed processing.

Here’s a code snippet that shows this:

```
Client client;
Token  token;

try {
    token.grab(client, false);    // Grab token non-exclusively.
    ... download large file ...
    token.release(client);
} catch (JSDTException e) {
    System.out.print(“Couldn’t download file.”);
}
```

Testing to see if all Clients had completed this download operation would be something like:

```
Token token;
```

```
try {
    while (token.test() != Token.NOT_IN_USE) {
              ... sleep or do something else ...
    }
    System.out.println(“Download completed for each Client.”);
} catch (JSDTException e) {
    System.out.print(“Couldn’t download file.”);
}
```

2.12 Manageable Objects

Sessions, Channels, ByteArrays and Tokens are all Manageable objects, and are subclassed from the Manageable class. They can all optionally have a Manager associated with them that authenticates each Client to see if they are allowed to do the requested operation (see Chapter 3, “Managers”).

A Manageable object can be used to:

- get the name of this Manageable object
- list the names of the Clients who are joined to this Manageable object
- determine if the Manageable object has a manager associated with it
- join this Manageable object
- leave this Manageable object
- enable or disable listener events (see Chapter 4, “Listeners”)
- enable or disable manager events (see Chapter 3, “Managers”)
- destroy the Manageable object
- invite a list of Clients to join this Manageable object
- expel a list of Clients from this Manageable object

2.12.1 *Getting a Manageable Objects Name*

Use the getName method to get a Manageable objects name. For example:

```
ByteArray byteArray;

System.out.println(“ByteArray name is: “ + byteArray.getName());
```

2.12.2 *Who’s Joined a Manageable Object*

Use the listClientNames method to list the names of all the Clients that are currently joined to this Manageable object. For example:

```
Session session;

try {
    String clientNames[] = session.listClientNames();
          if (clientNames == null) {
             System.out.println(“No Clients joined to Session.”);
         } else {
             System.out.println(clientNames.length + “ Clients
joined.”);
              for (int i = 0; i < clientNames.length; i++) {
                       System.out.println(clientNames[i]);
             }
    }
} catch (JSDTException e) {
    System.out.print(“Couldn’t list Client names.”);
}
```

2.12.3 *Joining a Manageable Object*

Use the join method to join a Manageable object. A Client needs to join that object before it can do most operations. A Client can also automatically join a Managable ByteArray, Channel or Token at the time of it’s creation.

If it is a managed object, then the Client is authenticated to determine if it is permitted to join.
For example, here’s some sample code for joining an unmanaged Token you already have a reference to:

```
Client client;
Token  token;

try {
    token.join(client);
} catch (JSDTException e) {
    System.out.print(“Couldn’t join Token.”);
}
```

2.12.4 *Leaving a Manageable Object*

Use the leave method on a Manageable object when you are no longer interested in it. Doing this will mean that that Client is no longer known to that Manageable object. JSDT applications should tidy themselves up before terminating, by leaving all objects they had previously joined.

For a Client that had previously joined a ByteArray and a Channel and the Session which contained them, this could look like:

```
Client    client;
Session   session;
ByteArray byteArray;
Channel   channel;

try {
    byteArray.leave(client);
    channel.leave(client);
    session.leave(client);
} catch (JSDTException e) {
    System.out.print(“Couldn’t leave successfully.”);
}
```

Leaving a Session will automatically cause the leave method to be called for that Client, for any Manageable object (ByteArray, Channel, Token) that that Client had joined within that Session.

2.12.5 *Customizing a Listeners Events*

Use the enableListenerEvents and disableListenerEvents methods, to customize which events you’d like a Listener to receive. This will reduce network traffic. Here’s some code that will setup a Session Listener to just receive join and leave events for that Session.

```
Session session;
SessionListener listener;
try {
    session.addSessionListener(listener);
    session.enableListenerEvents(listener,
                             SessionEvent.JOIN | Session.LEFT);
} catch (JSDTException e) {
    System.out.print(“Couldn’t setup Session Listener.”);
}
```

Note that the second parameter is a mask of the events you wish to enable.

2.12.6 *Customizing a Managers Events*

Use the enableManagerEvents and disableManagerEvents methods, to customize which events you’d like a Manager to do authentication on. Actions associated with events that are not in the Managers mask will occur without authentication. By default, a Manager will provide authentication for all actions associated with that Manageable object that need authenticating (see Chapter 3, “Managers”).

Only the creator of the managed object can call these two methods. Here’s some code that will setup a Session Manager to disable authentication for any ByteArrays and Channels that are created within this Session.

```
Session session;
SessionManager manager;
try {
    session.disableManagerEvents(manager,
                               SessionEvent.BYTEARRAY_CREATED |
                               SessionEvent.CHANNEL_CREATED);
} catch (JSDTException e) {
    System.out.print(“Couldn’t setup Session Manager mask.”);
}
```

Note that the second parameter is a mask of the events you wish to disable authentication on.

2.12.7 Destroying a Manageable Object

Use the destroy method to completely destroy any reference to a Manageable object. The server for that object will force all Clients to be expelled from it, before destroying all reference to it. When destroying a Session, the reference to that Session in the Registry is also removed.

Here’s a code snippet for destroying a Channel:

```
Channel channel;
Client  client;

try {
    channel.destroy(client);
} catch (JSDTException e) {
    System.out.print(“Couldn’t destroy Channel.”);
}
```

2.12.8 Inviting and Expelling Clients

Use the invite method to invite a list of Clients to join a Manageable object. Note that you need to have an array of Client objects in order to perform this task. Typically the only Client handles you have, are the ones you created yourself, so how do you do this?

It’s very simple. Use the special Clients that can be created by the ClientFactory.createClient method (see ClientFactory).

You will need to supply an object that has implemented the Client interface and an object that has implemented the ClientListener interface. The Client object will be used for authentication purposes and the ClientListener object will be notified when this special Client receives invitations to join a Session, ByteArray, Channel or Token, or when it has been expelled from a Session, ByteArray, Channel or Token. The same object could implement both the Client and the ClientListener interfaces.

Here’s some sample code that does exactly this for the socket implementation:

```
Client client;
URLString url = URLString.createClientURL(“myHost.Com”, 6677,
                                       “socket”, “FrankClient”);

try {
    client = new InviteClient(“Frank”);
    ClientFactory.createClient(client, url,
                               (ClientListener) client);
} catch (JSDTException e) {
    System.out.println(“Couldn’t create Client.”);
}
```

InviteClient looks like:

```
public class
InviteClient extends ClientAdaptor implements Client {

    private String name;

    public
    InviteClient(String name) {
              this.name = name;
    }

    public String
    getName() {
        return(name);
    }

    public Object
    authenticate(AuthenticationInfo info) {
        return(null);
    }

    public void
    sessionInvited(ClientEvent event) {
        String  resourceName = event.getResourceName();
        Session session;

        try {
            session = event.getSession();
```

```
                session.join(this);
            } catch (Exception e) {
                System.out.println(“Couldn’t join Session.”);
            }
        }
    }
```

As InviteClient extends ClientAdaptor, you just need to include in the methods for any Client events you are interested in. In this case, we are interested in the Session invite, so we have a sessionInvited method. When this is called, we get a handle to that Session, and then join it.

Here’s some code that will get a handle to this special Client from the Registry, and invite it to join a Session:

```
Client  inviteClient;
Session session;
URLString  url = URLString.createClientURL(“myHost.Com”, 6677,
                                        “socket”, FrankClient”);

try {
    Client[] clients = new Client[1];
    clients[0] = ClientFactory.lookupClient(url);
    session.invite(clients);
} catch (JSDTException e) {
    System.out.println(“Couldn’t invite Client to join Session.”);
}
```

You can use the expel method to expel a list of Clients from a Manageable object in a similar way. Client expulsion can only be done if the Manageable object has a manager associated with it, and can only be done by the creator of that managed object.

When a Client is expelled from a Manageable object, it is a forced expulsion , and there is nothing that that Clients application can do to stop it. Typical usage would be:

- if an application is running too slowly or not responding.
- if an application is doing something inappropriate.

2.13 Connection

If a JSDT proxy application performs an operation that requires sending a message to the server, and that server is not running, then a ConnectionException will be thrown. This failure indication only occurs as the result of attempting to send the message. Sometimes a JSDT proxy application would like to be asynchronously notified when the server is no longer running or responding. This can be used by adding a Connection listener using the Connection.addConnectionListener method (see Section 4.8, “Connection Listener).

A Connection can be used to:

- add or remove a Connection Listener
- set an implementation specific property (see Section 8.5, “Configurable Options”)

A Connection listeners connectionFailed method will be called when a connection failure is detected. It should do a Session.close call for every Session associated with that connection. This call will possibly thrown an exception which should be caught and ignored.

Here’s what a connectionFailed method might look like:

```
public void
connectionFailed(ConnectionEvent event) {
    try {
        mySession.close(false);
    } catch (JSDTException je) {
        System.err.println(“Ignoring exception.”);
    }
}
```

3 Managers

3.1 Overview

A JSDT Manager is an object which encapsulates some management policy for a Manageable object (i.e. a ByteArray, Channel, Session or Token). There can also be a manager associated with the Registry. There can only be one manager associated with a managed object. The manager must be assigned the very first time the managed object is created.

Access to a Session, ByteArray, Channel, Token or Registry can be controlled by assigning a Manager to it at creation time. The Manager will authenticate Clients wishing to perform a priviledged operation on this resource, and based upon their response will accept or reject them. The creation or destruction of ByteArrays, Channel and Tokens within a managed Session are examples of operations which require authentication.

The Manager of a ByteArray, Channel, Session or Token can invite a Client to join that resource using the Manageable.invite method. Clients then join that resource using the regular join method. In a similar way, a Manager can force a Client to leave a resource with the Manageable.expel method.

3.2 Authentication

A special AuthenticationInfo class is used to encapsulate all the information needed by a Client to determine what they are being asked to authenticate.

JSDT applications implement the Client interface in order to join the various JSDT objects, and send and receive data. One of the two methods that the Client interface declares is:

```
public Object authenticate(AuthenticationInfo info);
```

Authentication takes place in the following situations:

- when a Client tries to create or destroy a Session or Client in a managed Registry
- when a Client tries to create or destroy a ByteArray in a managed Session
- when a Client tries to create or destroy a Channel in a managed Session
- when a Client tries to create or destroy a Token in a managed Session
- when a Client tries to join a managed Session
- when a Client tries to join a managed ByteArray
- when a Client tries to join a managed Channel
- when a Client tries to join a managed Token

The manager sends the Client an authentication request. Within this request is a challenge. The Client replies with a response. This response is validated by the manager and determines if the Client will be allowed to do the requested operation.

The challenge given by the manager and the response provided by the Client are both Java objects. There must be some agreed policy between the manager and the Client with regards to these objects. In other words the Client needs to know what to do with the challenge and how to respond to it, and the manager needs to know how to handle that response.

The AuthentificationInfo object contains the following information:

- the Session associated with this authentication operation (null for Registry authentication)
- the type of managed object. This will be one of:
  - AuthenticationInfo.REGISTRY
  - AuthentificationInfo.BYTEARRAY
  - AuthentificationInfo.CHANNEL
  - AuthentificationInfo.SESSION
  - AuthentificationInfo.TOKEN
- the name of the object associated with this authentication operation. This will be the name of the ByteArray, Channel, Session or Token being created, or the name of the ByteArray, Channel, Session or Token being destroyed, or the name of the manageable object the Client is trying to join
- the authentication action, This will be one of:
  - AuthentificationInfo.JOIN
  - AuthentificationInfo.CREATE_BYTEARRAY
  - AuthentificationInfo.DESTROY_BYTEARRAY
  - AuthentificationInfo.CREATE_CHANNEL
  - AuthentificationInfo.DESTROY_CHANNEL
  - AuthenticationInfo.CREATE_SESSION
  - AuthenticationInfo.DESTROY_SESSION
  - AuthentificationInfo.CREATE_TOKEN
  - AuthentificationInfo.DESTROY_TOKEN

- the challenge given by the manager

3.3 *Session Manager*

A Session Manager is associated with a Session at Session create time. There is a variant of the SessionFactory.createSession method to do this:

```
public static Session
createSession(String url, SessionManager sessionManager)
throws NoRegistryException, NoSuchHostException,
       InvalidURLException, NoSuchSessionException,
            ManagerExistsException, TimedOutException;
```

A Session Manager needs to implement the SessionManager interface which declares one method:

```
public boolean
sessionRequest(Session session,
               AuthenticationInfo info, Client client);
```

Here is the simplest form of a class which implements SessionManager:

```
public class
ExampleSessionManager implements SessionManager {

    public boolean
    sessionRequest(Session session,
                                  AuthenticationInfo info, Client
client) {
            String challenge = “<challenge>”;
       String expectedResponse = “<response>”;
       String reply = null;

       info.setChallenge(challenge);
       reply = (String) client.authenticate(info);
       return(reply.equals(expectedResponse));
    }
}
```

What you would put in the Session Manager’s sessionRequest method is given in the example code below.

Here’s a snippet of code showing the creation of a Session Manager and it’s association to a Session at Session creation time:

```
SessionManager sessionManager;
```

```
Session       session;
String url =
           "jsdt://stard:4461/socket/Session/managedSession";

try {
    sessionManager = new ExampleSessionManager();
    session       = SessionFactory.createSession(url,
                                          sessionManager);
} catch (JSDTException e) {
    System.out.print("Couldn't create Session with manager.");
}
```

A Client could now attempt to join this managed Session:

```
Session session;
Client  client;

try {
    session.join(client);
} catch (JSDTException e) {
    System.out.print("Couldn't join the managed Session.");
}
```

This attempt would cause the Session Managers sessionRequest method to be called. This could look something like:

```
public boolean
sessionRequest(Session session,
               AuthenticationInfo info, Client client) {
    String reply = null;
    info.setChallenge("ABCDEF");
    reply = (String) client.authenticate(info);
    return(reply.equals("abcdef"));
}
```

Note that this manager is sending the Client a challenge of "ABCDEF". The Client needs to reply with "abcdef" to be successfully authenticated, and allowed to join the Session. It's authenticate method could look something like this:

```
public Object
authenticate(AuthenticationInfo info) {
    int    action    = info.getAction();
         String type      = info.getType();
         String name      = info.getName();
         String challenge = (String) info.getChallenge();
    String response  = null;

    if (action == AuthenticationInfo.JOIN &&
        type  == AuthentificationInfo.SESSION &&
        name.equals("managedSession") &&
        challenge.equals(“ABCDEF”)) {
        response = “abcdef”;
    } else {
        ... process other authentication requests ...
    }
    return(response);
}
```

3.4 Channel Manager

A Channel Manager is associated with a Channel at Channel create time. There is a variant of the Session.createChannel method to do this:

```
public Channel
createChannel(Client client, String channelName,
              boolean reliable, boolean ordered,
              ChannelManager channelManager)
throws NoSuchSessionException, NoSuchClientException,
       NoSuchHostException, PermissionDeniedException,
       ManagerExistsException, TimedOutException;
```

A Channel Manager needs to implement the ChannelManager interface which declares one method:

```
public boolean
channelRequest(Channel channel,
               AuthenticationInfo info, Client client);
```

Here is the simplest form of a class which implements ChannelManager:

```
public class
ExampleChannelManager implements ChannelManager {

    public boolean
    channelRequest(Channel channel,
                   AuthenticationInfo info, Client client) {
        boolean validation;

        ... handle channel authentication request ...
        return(validation);
    }
}
```

See Session Manager” for an example of authentication between a Client and the Manager of a managed object.

3.5 ByteArray Manager

A ByteArray Manager is associated with a ByteArray at ByteArray create time. There are two variants of the Session.createByteArray method to do this:

```
public ByteArray
createByteArray(Client client, String byteArrayName,
                byte[] value,
                ByteArrayManager byteArrayManager)
throws NoSuchSessionException, NoSuchClientException,
       NoSuchHostException, PermissionDeniedException,
       ManagerExistsException, TimedOutException;

public ByteArray
createByteArray(Client client, String byteArrayName,
                byte[] value, int offset, int length,
                ByteArrayManager byteArrayManager)
throws NoSuchSessionException, NoSuchClientException,
       NoSuchHostException, PermissionDeniedException,
       ManagerExistsException, TimedOutException;
```

A ByteArray Manager needs to implement the ByteArrayManager interface which declares one method:

```
public boolean
byteArrayRequest(ByteArray byteArray,
                 AuthenticationInfo info, Client client);
```

Here is the simplest form of a class which implements ByteArrayManager:

```
public class
ExampleByteArrayManager implements ByteArrayManager {

    public boolean
    byteArrayRequest(ByteArray byteArray,
                     AuthenticationInfo info, Client client) {
        boolean validation;

        ... handle bytearray authentication request ...
        return(validation);
    }
}
```

See Session Manager" for an example of authentication between a Client and the Manager of a managed object.

3.6 Token Manager

A Token Manager is associated with a Token at Token create time. There is a variant of the Session.createToken method to do this:

```
public Token
createToken(Client client, String tokenName,
            TokenManager tokenManager)
throws NoSuchSessionException, NoSuchClientException,
       NoSuchHostException, PermissionDeniedException,
       ManagerExistsException, TimedOutException;
```

A Token Manager needs to implement the TokenManager interface which declares one method:

```
public boolean
tokenRequest(Token token,
             AuthenticationInfo info, Client client);
```

Here is the simplest form of a class which implements TokenManager:

```
public class
ExampleTokenManager implements TokenManager {

    public boolean
    tokenRequest(Token token,
                 AuthenticationInfo info, Client client) {
        boolean validation;

        ... handle token authentication request ...
        return(validation);
    }
}
```

See Session Manager” for an example of authentication between a Client and the Manager of a managed object.

3.7 Registry Manager

A Registry Manager is associated with a Registry at Registry create time. There is a variant of the RegistryFactory.startRegistry method to do this:

```
public static void
startRegistry(String registryType, RegistryManager registryManager)
```

```
     throws RegistryExistsException, NoRegistryException,
            ManagerExistsException, PermissionDeniedException;
```

A Registry Manager needs to implement the RegistryManager interface which declares one method:

```
public boolean
registryRequest(AuthenticationInfo info, Client client);
```

Here is the simplest form of a class which implements RegistryManager:

```
public class
ExampleRegistryManager implements RegistryManager {

    public boolean
    registryRequest(AuthenticationInfo info, Client client) {
        boolean validation;

        ... handle registry authentication request ...
        return(validation);
    }
}
```

See Session Manager" for an example of authentication between a Client and the Manager of a managed object.

4 Listeners

4.1 Overview

A Listener is an object that has registered its interest in being notified about changes in state of some other given JSDT object. It can listen for changes in the state of a Session, Channel, ByteArray, Token, Client or Registry.

A Session Listener will be notified about Clients joining, leaving, being invited to join, or being expelled from a Session. It will also be notified when a ByteArray, Channel or Token is created or destroyed within that Session, and when a Session is destroyed.

A Channel Listener will be notified about Clients joining, leaving, being invited to join or being expelled from a Channel. It will also be notified when a ChannelConsumer has been added or removed from a Channel.

A ByteArray Listener will be notified about Clients joining, leaving, being invited to join or being expelled from a ByteArray.

A Token Listener will be notified about Clients joining and leaving a Token, plus a Client being invited to join a Token or expelled from a Token. It will also be notified when a Client has given or requested or grabbed or released a Token.

A Client Listener will be notified when it has been invited to join or been expelled from a ByteArray, Channel, Session or Token. It will also be notified when a Client is given a Token.

A Registry Listener will be notified when a Session or Client is created or destroyed within the Registry, or if the underlying connection to the Registry has failed.

A Connection Listener will be notified when an underlying connection failure has occurred.
Listeners, Events and Adaptors use another JSDT package:

```
com.sun.media.jsdt.event.*;
```

When writing any code that makes use of these three kinds of objects, make sure you include an import line for this package.

4.2 *Session Listener*

A Session Listener can be associated with a Session after the Session has been created. The Session.addSessionListener method is used to achieve this.

A Session Listener needs to implement the SessionListener interface which declares ten methods:

```
public void byteArrayCreated(SessionEvent event);
public void byteArrayDestroyed(SessionEvent event);
public void channelCreated(SessionEvent event);
public void channelDestroyed(SessionEvent event);
public void sessionDestroyed(SessionEvent event);
public void sessionJoined(SessionEvent event);
public void sessionLeft(SessionEvent event);
public void sessionInvited(SessionEvent event);
public void sessionExpelled(SessionEvent event);
public void tokenCreated(SessionEvent event);
public void tokenDestroyed(SessionEvent event);
```

These are a lot of methods to implement if you are only interested in certain event types (joining and leaving the Session for example). For an alternate approach to Session event notification, See Chapter 6, “Adaptors”.

4.3 *Channel Listener*

A Channel Listener can be associated with a Channel after the Channel has been created. The Channel.addChannelListener method is used to achieve this. A Channel Listener needs to implement the ChannelListener interface which declares four methods:

```
public void channelJoined(ChannelEvent event);
public void channelLeft(ChannelEvent event);
public void channelInvited(ChannelEvent event);
public void channelExpelled(ChannelEvent event);
public void channelConsumerAdded(ChannelEvent event);
public void channelConsumerRemoved(ChannelEvent event);
```

These are a lot of methods to implement if you are only interested in certain event types (joining and leaving the Channel for example). For an alternate approach to Channel event notification, see Chapter 6, “Adaptors”.

4.4 *ByteArray Listener*

A ByteArray Listener can be associated with a ByteArray after the ByteArray has been created. The ByteArray.addByteArrayListener method is used to achieve this.

A ByteArray Listener needs to implement the ByteArrayListener interface which declares five methods:

```
public void byteArrayJoined(ByteArrayEvent event);
public void byteArrayLeft(ByteArrayEvent event);
public void byteArrayValueChanged(ByteArrayEvent event);
public void byteArrayInvited(ByteArrayEvent event);
public void byteArrayExpelled(ByteArrayEvent event);
```

These are a lot of methods to implement if you are only interested in certain event types (when the byte array value changes for example). For an alternate approach to ByteArray event notification, see Chapter 6, “Adaptors”.

4.5 *Token Listener*

A Token Listener can be associated with a Token after the Token has been created. The Token.addTokenListener method is used to achieve this. A Token Listener needs to implement the TokenListener interface which declares eight methods:

```
public void tokenJoined(TokenEvent event);
public void tokenLeft(TokenEvent event);
public void tokenGiven(TokenEvent event);
public void tokenRequested(TokenEvent event);
public void tokenGrabbed(TokenEvent event);
public void tokenReleased(TokenEvent event);
public void tokenInvited(TokenEvent event);
public void tokenExpelled(TokenEvent event);
```

These are a lot of methods to implement if you are only interested in certain event types (joining and leaving for example). For an alternate approach to Token event notification, see Chapter 6, “Adaptors”.

4.6 Client Listener

A Client Listener can be associated with a Client after the Client has been created. The Client needs to implement the ClientListener interface which declares eight methods:

```
public void byteArrayInvited(ClientEvent event);
public void byteArrayExpelled(ClientEvent event);
public void channelInvited(ClientEvent event);
public void channelExpelled(ClientEvent event);
public void sessionInvited(ClientEvent event);
public void sessionExpelled(ClientEvent event);
public void tokenInvited(ClientEvent event);
public void tokenExpelled(ClientEvent event);
public void tokenGiven(ClientEvent event);
```

These are a lot of methods to implement if you are only interested in certain event types (session invites for example). For an alternate approach to Client event notification, see Chapter 6, "Adaptors".

4.7 Registry Listener

A Registry Listener can be associated with a Registry after the Registry has been started. The RegistryFactory.addRegistryListener method is used to achieve this. A Registry Listener needs to implement the RegistryListener interface which declares five methods:

```
public void sessionCreated(RegistryEvent event);
public void sessionDestroyed(RegistryEvent event);
public void clientCreated(RegistryEvent event);
public void clientDestroyed(RegistryEvent event);
public void connectionFailed(RegistryEvent event);
```

These are a lot of methods to implement if you are only interested in certain event types (session creates for example). For an alternate approach to Registry event notification, see Chapter 6, "Adaptors".

4.8 Connection Listener

A Connection Listener can be associated with a Connection to a particular host. The Connection.addConnectionListener method is used to achieve this. A Connection Listener needs to implement the ConnectionListener interface which declares one method:

```
public void connectionFailed(ConnectionEvent event);
```

5 Events

5.1 Overview

Events encapsulate a change of state in a JSDT object. They are sent to Listeners who have registered interest in such state changes. Convenience methods are available to extract that information and handle it appropriately.

5.2 Session Event

Session events are created for the following actions:

- when a ByteArray has been created
- when a ByteArray has been destroyed
- when a Channel has been created
- when a Channel has been destroyed
- when a Token has been created
- when a Token has been destroyed
- when a Client has joined a Session
- when a Client has left a Session
- when a Client has been invited to join a Session
- when a Client has been expelled from a Session
- when a Session has been destroyed

A Session Event contains the following information:

- the type of this Session event. Valid types are:
    - SessionEvent.BYTEARRAY_CREATED
    - SessionEvent.BYTEARRAY_DESTROYED
    - SessionEvent.CHANNEL_CREATED
    - SessionEvent.CHANNEL_DESTROYED
    - SessionEvent.TOKEN_CREATED
    - SessionEvent.TOKEN_DESTROYED

  - SessionEvent.JOINED
  - SessionEvent.LEFT
  - SessionEvent.INVITED
  - SessionEvent.EXPELLED
  - SessionEvent.DESTROYED
- the session associated with this Session event
- the name of the Client causing this event
- the name of the resource within the Session that the event affects

5.3 Channel Event

Channel events are created for the following actions:

- when a Client has joined a Channel
- when a Client has left a Channel
- when a Client has been invited to join a Channel
- when a Client has been expelled from a Channel
- when a Client has added a Consumer to the Channel
- when a Client has removed a Consumer from the Channel

A Channel Event contains the following information:

- the type of this Channel event. Valid types are:
  - ChannelEvent.JOINED
  - ChannelEvent.LEFT
  - ChannelEvent.INVITED
  - ChannelEvent.EXPELLED
  - ChannelEvent.CONSUMER_ADDED
  - ChannelEvent.CONSUMER_REMOVED
- the session associated with this Channel event
- the Channel associated with this Channel event
- the name of the Client causing this event

5.4 ByteArray Event

ByteArray events are created for the following actions:

- when a Client has joined a ByteArray
- when a Client has left a ByteArray
- when the value of a ByteArray changes
- when a Client has been invited to join a ByteArray

- when a Client has been expelled from a ByteArray

A ByteArray Event contains the following information:

- the type of this ByteArray event. Valid types are:
  - ByteArrayEvent.JOINED
  - ByteArrayEvent.LEFT
  - ByteArrayEvent.VALUE_CHANGED
  - ByteArrayEvent.INVITED
  - ByteArrayEvent.EXPELLED
- the session associated with this ByteArray event
- the ByteArray associated with this ByteArray event
- the name of the Client causing this event

5.5 *Token Event*

Token events are created for the following actions:

- when a Token has been given from one Client to another
- when a Client has grabbed a Token
- when a Client has inhibited a Token
- when a Client has joined a Token
- when a Client has left a Token
- when a Client has released itself from a Token
- when a Client has requested a Token
- when a Client has been invited to join a Token
- when a Client has been expelled from a Token

A Token Event contains the following information:

- the type of this Token event. Valid types are:
  - TokenEvent.GIVEN
  - TokenEvent.GRABBED
  - TokenEvent.INHIBITED
  - TokenEvent.JOINED
  - TokenEvent.LEFT
  - TokenEvent.RELEASED
  - TokenEvent.REQUESTED
  - TokenEvent.INVITED
  - TokenEvent.EXPELLED
- the session associated with this Token event
- the Token associated with this Token event

- the name of the Client causing this event

5.6 *Client Event*

Client events are created for the following actions:

- when a Client has been invited to join a ByteArray
- when a Client has been expelled from a ByteArray
- when a Client has been invited to join a Channel
- when a Client has been expelled from a Channel
- when a Client has been invited to join a Session
- when a Client has been expelled from a Session
- when a Client has been invited to join a Token
- when a Client has been expelled from a Token
- when a Client has been given a Token

A Client Event contains the following information:

- the type of this Client event. Valid types are:
  - ClientEvent.BYTEARRAY_INVITED
  - ClientEvent.BYTEARRAY_EXPELLED
  - ClientEvent.CHANNEL_INVITED
  - ClientEvent.CHANNEL_EXPELLED
  - ClientEvent.SESSION_INVITED
  - ClientEvent.SESSION_EXPELLED
  - ClientEvent.TOKEN_INVITED
  - ClientEvent.TOKEN_EXPELLED
  - ClientEvent.TOKEN_GIVEN
- the session associated with this Client event
- the Client associated with this Client event
- the name of the resource that this event occurred on

5.7 *Registry Event*

Registry events are created for the following actions:

- when a Session has been created
- when a Session has been destroyed
- when a Client has been created
- when a Client has been destroyed
- when the underlying connection to the Registry has failed

A Registry Event contains the following information:

- the type of this Registry event. Valid types are:
    - RegistryEvent.SESSION_CREATED
    - RegistryEvent.SESSION_DESTROYED
    - RegistryEvent.CLIENT_CREATED
    - RegistryEvent.CLIENT_DESTROYED
    - RegistryEvent.CONNECTION_FAILED
- the name of the Client causing this event
- the URL string for the resource within the Registry that this event affects
- the host address of the Registry
- the port number the Registry is running on, on that host

5.8 Connection Event

Connection events are created for the following actions:

- when a connection failure has occurred

A Connection Event contains the following information:

- the type of this Connection event. Valid types are:
    - ConnectionEvent.CONNECTION_FAILED
- the host address that caused this event
- the port number on that host that caused this event

6 Adaptors

6.1 Overview

Adaptors are abstract classes that you can use to avoid having to provide empty method implementations for all the Listener methods you are not interested in. Instead of implementing the interface for the Listener you are interested in, you extend its Adaptor instead, filling in only the methods for the event types you want to handle. The Adaptor silently handles the remainder.

A SessionAdaptor can be used to receive all Session Events.

A ChannelAdaptor can be used to receive all Channel Events.

A ByteArrayAdaptor can be used to receive all ByteArray Events.

A TokenAdaptor can be used to receive all Token Events.

A ClientAdaptor can be used to receive all Client Events.

A RegistryAdaptor can be used to receive all Registry Events.

A ConnectionAdaptor can be used to receive all Connection Events.

A couple of examples should show how easy Adaptors are to use.

6.2 Handling ByteArray Value Changes

A ByteArray Listener needs to implement methods to handle when a Client joins, leaves, is invited to join, or is expelled from a ByteArray, plus when the value of a ByteArray changes. Five methods in all. Perhaps you only care when the value of the ByteArray has changed, so you can do some local change (such as redisplay the new stock information on the users screen).

Here’s how to use a ByteArrayAdaptor to achieve this in a simple manner.

All we need to do is create a class that extends ByteArrayAdaptor and contains a byteArrayValueChanged method which we fill out to handle our needs:

```
import com.sun.media.jsdt.*;
import com.sun.media.jsdt.event.*;

public class
ExampleByteArrayAdaptor extends ByteArrayAdaptor {

    public void
    byteArrayValueChanged(ByteArrayEvent event) {
        try {
            byte[] newValue = event.getByteArray().getValue();
                       ... display new stock information ...
        } catch (NoSuchByteArrayException noe) {
                       System.out.println(“Couldn’t get stock info.”);
        }
    }
}
```

All the other ByteArray event types are handled silently by the empty methods in the ByteArrayAdaptor class.

6.3 *Giving a Token*

Using a TokenAdaptor and a ClientAdaptor can simplify the code needed for giving a Token from one Client to another. Lets assume the giver and the receiver have already successfully created and joined the unmanaged Session where the Token exchange is going to occur.

First the giver will create the Token and join it. It will add a Token Listener to that Token, then grab the Token:

```
Client  giverClient;
Session session;
Token   token;

try {
    giverClient = new GiverClient(“Giver”);
    token = session.createToken(giverClient, “TheToken”, true);
    token.addTokenListener((TokenListener) giverClient);
     token.grab(giverClient, true);
 } catch (JSDTException e) {
     System.out.println(“Giver: Couldn’t setup Token.”);
 }
```

We’ve used a class called GiverClient with these operations. GiverClient implements Client and extends TokenAdaptor. In this class we are only interested in when other Clients join the Token, so we only implement the tokenJoined method. All other events are silently handled by the Token   Adaptor. When a Client named “Receiver” joins the Token we can then give it to that Client. Here’s what GiverClient looks like:

```
public class
GiverClient extends TokenAdaptor implements Client {

    protected String name;

    public
    GiverClient(String name) {
        this.name = name;
    }

    public Object
    authenticate(AuthenticationInfo info) {
        return(null);
    }

    public String
    getName() {
        return(name);
    }

    public void
    tokenJoined(TokenEvent event) {
        String clientName = event.getClientName();
        Token  token      = event.getToken();

       if (clientName.equals(“Receiver”)) {
            try {
                token.give(this, clientName);
            } catch (JSDTException e) {
                System.out.println(“Couldn’t give Token.”);
            }
        }
    }
}
```

Now, on the receiving side we need to do something similar. The receiver   will create the Token and join it:

```
Client  receiverClient;
Session session;
Token   token;
```

```
try {
    receiverClient = new ReceiverClient(“Receiver”);
     token = session.createToken(receiverClient, “TheToken”, true);
} catch (JSDTException e) {
    System.out.println(“Receiver: Couldn’t setup Token.”);
}
```

The receiver Client is slightly different to the giver Client. The ReceiverClient class extends ClientAdaptor and implements Client. In this case we are only interested in when another Client gives the Token specifically to us, so we only implement the tokenGiven(ClientEvent event) method. All other Client events are silently handled by the Client Adaptor.

Note that extending a TokenAdaptor adding ourselves as a Token listener and implementing the tokenGiven(TokenEvent event) method would tell us when Tokens were being given, but would not tell us that the Token was being given to us specifically.

When we are given the Token we can then grab it. Here’s what ReceiverClient looks like:

```
public class
ReceiverClient extends ClientAdaptor implements Client {

    private String name;

    public
    ReceiverClient(String name) {
        this.name = name;
    }

    public Object
    authenticate(AuthenticationInfo info) {
        return(null);
    }

    public String
    getName() {
        return(name);
    }

    public void
    tokenGiven(ClientEvent event) {
        String tokenName = event.getResourceName();

        if (tokenName.equals(token.getName())) {
            try {
                token.grab(this, true);
            } catch (JSDTException e) {
                System.out.println(“Couldn’t grab Token.”);
            }
        }
    }
}
```

7 Exceptions

7.1 Overview

JSDT throws various exceptions when an error has occured. It is up to the JSDT application program to catch these and handle them appropriately.

All JSDT Exceptions are derived from the JSDTException class.

7.2 Exception Types

The following exceptions can be thrown:

- AlreadyBoundException - thrown when a Session or Client with this URL is already bound in the Registry.
- ClientNotGrabbingException - thrown to indicate that an attempt has been made to release a Token that had not been previously grabbed.
- ClientNotReleasedException - thrown when an attempt is made to exclusively grab a Token that was still being grabbed by another Client.
- ConnectionException - thrown when some kind of network error has occurred when two components within a JSDT collaboration have failed to communicate with each other.
- InvalidClientException - thrown when an attempt is made to access a Client that is invalid. The most likely reason for this is that its getName method returns null.
- InvalidURLException - thrown when an attempt was made to use a URL with the SessionFactory or Naming classes, which is not in the required format.
- ManagerExistsException - thrown when an attempt is made to create a managed ByteArray, Channel or Token which already exists, and which already has a manager associated with it.
- NameInUseException - thrown when an attempt is made to use a JSDT object which already has this name.
- NoRegistryException - thrown when an attempt is made to contact the JSDT Registry, and it is not running. There should be a Registry running on every machine that is serving up JSDT Sessions or Clients.

- NoSuchByteArrayException - thrown when an attempt is made to access a ByteArray that doesn’t exist.
- NoSuchChannelException - thrown when an attempt is made to access a Channel that doesn’t exist.
- NoSuchClientException - thrown when an attempt is made to access a Client that doesn’t exist.
- NoSuchConsumerException - thrown when an attempt is made to access a ChannelConsumer that doesn’t exist.
- NoSuchHostException - thrown when an attempt is made to access a remote host that doesn’t exist.
- NoSuchListenerException - thrown when an attempt is made to access a Listener that doesn’t exist.
- NoSuchManagerException - thrown when an attempt is made to access a Manager that doesn’t exist.
- NoSuchSessionException - thrown when an attempt is made to access a Session that doesn’t exist.
- NoSuchTokenException - thrown when an attempt is made to access a Token that doesn’t exist.
- NotBoundException - thrown when an attempt is made to access a JSDT Session or Client that is not bound with the Registry.
- PermissionDeniedException - thrown when an attempt is made to do an operation on a JSDT object when it’s not permissible.
- PortInUseException - thrown when an attempt is made to use a port that is already being used by another application.
- RegistryExistsException - thrown when an attempt is made to start a Registry when there is one already running.
- TimedOutException - thrown if no reply was received for this operation in the given timeout period.

8 Implementations

8.1 Overview

The design of JSDT is independent of the underlying implementation. Nothing is specified on how the various components of JSDT applications communicate with each other. This is left upto the individual implementations. The *com.sun.media.jsdt* classes dynamically load the implementation the user requires, based on the *<impl>* field of the JSDT Session URL Strings.

This release currently comes with two implementations. These are described in more detail in this chapter, and include information that is specific to that implementation.

8.2 Socket

The *socket* implementation uses TCP sockets to send messages to communicate between the collaborating JSDT applications. It uses UDP sockets to provide unreliable Channels. It will keep sockets open continually where possible.

You can also provide alternate sockets using this implementation (see socketFactoryClass).

8.2.1 Limitations

- The maximum size of the byte array, that can be sent in a Data object over unreliable Channels is just less than 8 Kbytes, due to an underlying limitation in the size of UDP packets.
- Data priorities are ignored.

8.2.2 SSL Support

Support has been added for SSL sockets. To enable this support, you will need to add two lines near the very beginning of your JSDT applications:

```
Connection.setProperty("socketFactoryClass",

"com.sun.media.jsdt.socket.SSLSocketFactory");
Connection.setProperty("SSLKeyStore", “/path/to/keystorefile”);
```

See also socketFactoryClass and SSLKeyStore.

8.3 HTTP

The http implementation try to use a direct TCP socket connection wherever possible. If this connection attempt fails, it will try to use HTTP POST commands to send messages from the various JSDT proxy applications to the JSDT server application, and the Registry. After sending each JSDT message, and getting it’s reply the connection is closed. The various asynchronous messages that JSDT can send are handled by storing them on the server for each proxy, until that proxy pings the server, checking if there are any messages for it.

Important Note.
When terminated, JSDT applications that use the HTTP implementation need to properly cleanup their JSDT resources. As there is no permanent connection between either the proxies and the server or the server and the Registry, this doesn’t automatically happen.

8.3.1 Limitations

- No unreliable Channels.
- Data priorities are ignored.

8.3.1 Working through firewalls

If the HTTP implementation of JSDT cannot get a direct TCP socket connection, it uses HTTP-tunneling, in a similar manner to the way RMI does it. This method is popular since it requires almost no setup, and works quite well in firewalled environments which permit handle HTTP through a proxy, but disallow regular outbound TCP connections.

An attempt will be made to tunnel JSDT requests through that proxy server, one at a time.

There are two forms of HTTP-tunneling, tried in order. The first is http-to-port; the second is http-to-cgi.

In http-to-port tunneling, JSDT attempts a HTTP POST request to a http: URL directed at the exact hostname and port number of the target server. The HTTP request contains a single JSDT request. If the HTTP proxy accepts this URL, it will forward the POST request to the listening JSDT server, which will recognise the request and unwrap it. The result of the call is wrapped in a HTTP reply, which is returned through the same proxy.

Often, HTTP proxies will refuse to proxy requests to unusual port numbers. In this case, JSDT will fall back to http-to-cgi tunneling. The JSDT request is encapsulated in a HTTP POST request as before, but the request URL is of the form http://hostname:80/cgi-bin/java-jsdt.cgi?port=n (where hostname and n are the hostname and port number of the intended JSDT server). There must be a HTTP server listening on port 80 on the server host, which will run the java-jsdt.cgi script (supplied with the JSDT distribution), which will in turn forward the request to a JSDT server listening on port n. JSDT can unwrap a HTTP-tunneled request without help from a http server, CGI script, or any other external entity. So, if the client's HTTP proxy can connect directly to the server's port, then you don't need a java-jsdt.cgi script at all.

Note that you can set an alternate HTTP proxy port using the httpTunnelPort property (see httpTunnelPort).

If you are running a web server that is capable of running Java servlets, then you can alias the "/cgi-bin/java-jsdt.cgi" script to a Java servlet class called com.sun.media.jsdt.http.ServletHandler which is also included with the JSDT distribution. This is much faster that a CGI script.

Note that the http-to-cgi method opens a dramatic security hole on the server side, since without modification it will redirect any incoming request to any port.

For more details on how to setup the JSDT HTTP implementation to work through firewall, see the section in the JSDT Release Notes.

8.4 Configurable Options

The following properties are available to allow you to adjust the way each implementation operates. Add the given line of code to your JSDT application. For each option, an indication is given for which implementations it is used.

8.4.1 alwaysPing

[http]

```
Connection.setProperty("alwaysPing", "true");
```

This indicates whether we should always ping for asynchronous messages irrespective of the socket factory class being used.

8.4.2 cleanupPeriod

[http]

```
Connection.setProperty("cleanupPeriod", "20000");
```

The number of milliseconds to wait before cleaning up a pinging client, that hasn't pinged. The default is 30000 (30 seconds).

8.4.3 cleanupPingingClients

[http]

```
Connection.setProperty("cleanupPingingClients", "false");
```

This property indicates whether we should cleanup pinging Clients, if we haven't heard from them, in cleanupPeriod milliseconds. By default, this property is set to true.

8.4.4 debugStream

[all]

```
Connection.setProperty("debugStream", "System.out");
```

This property determines the PrintStream used for the output of debug and error messages. By default, this value is System.err.

8.4.5 giveTime

[all]

```
Connection.setProperty("giveTime", "10000");
```

This indicates the period in time (in milliseconds), that a Token.give operation will be given to complete. If the Token has not been successfully given during this period, it’s ownership will revert to the original giver, and it will no longer be in the TokenEvent.GIVEN state. By default, this value is set to 15000 (15 seconds).

8.4.6 httpFactoryClass

[http]

```
Connection.setProperty(“httpFactoryClass”,
        “com.sun.media.jsdt.http.HttpToPortSocketFactory”);
```

This allows you to specify an alternate factory class for creating sockets. This class file must implement the com.sun.media.jsdt.impl.JSDTSocketFactory interface, which defines two methods:

```
Socket
createSocket(String address, int port)
            throws IOException, UnknownHostException;

ServerSocket
createServerSocket(int port) throws IOException;
```

To just use TCP sockets, this can be set to:

```
com.sun.media.jsdt.http.TCPSocketFactory
```

To just use direct HTTP sockets, ths can be set to:

```
com.sun.media.jsdt.http.HttpToPortSocketFactory
```

To just use tunneling HTTP sockets, this can be set to:

```
com.sun.media.jsdt.http.HttpToCGISocketFactory
```

By default, the `com.sun.media.jsdt.http.JSDTMasterSocketFactory` class is used which first tries to establish a direct TCP connection. If this fails, it then tries to use an HTTP connection to a direct port. In this also fails, it falls back to trying to tunnel an HTTP connection through the firewall.

8.4.7 httpTunnelPort

[http]

```
Connection.setProperty(“httpTunnelPort”, “8080”);
```

The port number of the web server which is running a CGI script or a Java Servlet, that will “tunnel” HTTP requests through a firewall from proxies to the server, and back. By default, this value is set to 80.

8.4.8 httpURIName

[http]

```
Connection.setProperty(“httpURIName”, “/jsdtserver12”);
```

In a typical commercial web server configuration, the web site is front-ended by a redirection proxy web server such as those available from Netscape or Apache. The rediection proxy redirects all the received HTTP requests to a back end web server or application server depending on the URL requested.

The httpURIName propertyl is designed to assist in configuring redirection proxies.. The default URI field in the POST request ("/") may be changed to any value.

For example httpURIName could be set to "/jsdtserver12" with a corresponding entry in the redirection filter table of

"http://sun.com/jsdtserver12":"jsdtserver12.sun.com".

This will force the redirection proxy to automatically redirect
all JSDT requests it receives to the machine jsdtserver12.

8.4.9 keepAlivePeriod

[all]

```
Connection.setProperty(“keepAlivePeriod”, “10000”);
```

This defines the number of milliseconds to wait before pinging the server to see if it’s still alive. By default, this value is set to 5000 (five seconds).

8.4.10 maxQueueSize

[all]

```
Connection.setProperty(“maxQueueSize”, “30”);
```

This defines the maximum number of incoming messages a connection will store, before it waits for the queue to be emptied. By default, this value is set to 15.

8.4.11 maxThreadPoolSize

[all]

```
Connection.setProperty(“maxThreadPoolSize”, “15”);
```

This defines the maximum number of threads that will be used to handle incoming Data received over Channels. By default, this value is set to 5.

8.4.12 pingPeriod

[http]

```
Connection.setProperty(“pingPeriod”, “250”);
```

The defines the period (in milliseconds) used by the each JSDT application to ping for any asynchronous messages that might be queued for that connection. By default, this value is set to 500.

Note that if you have a direct HTTP port connection, with a permanent connection established, than pinging is not done. It is only needed if the HTTP connection was forced to revert to tunneling and is talking to a CGI script or a Java servlet running on the server, or if the permanent connection was closed for some reason.

8.4.13 registryPingPeriod

[http]

```
Connection.setProperty(“registryPingPeriod”, “2000”);
```

The defines the period (in milliseconds) used by the each JSDT application to ping for any asynchronous messages from the Registry, that might be queued for that connection. By default, this value is set to 1000. This is only needed if you have one or more RegistryListeners.

Note that if you have a direct HTTP port connection, with a permanent connection established, than pinging is not done. It is only needed if the HTTP connection was forced to revert to tunneling and is talking to a CGI script or a Java servlet running on the server, or if the permanent connection was closed for some reason.

8.4.14 registryPort

[all]

```
Connection.setProperty(“registryPort”, “8000”);
```

The port number that the Registry should run on. By default this is 4561.

8.4.15 registryTime

[all]

```
Connection.setProperty(“registryTime”, “30”);
```

This defines the period (in seconds) that the RegistryFactory.startRegistry method will wait for the Registry to start. If it hasn’t started during this period, then a NoRegistryException exception is thrown. By default, this value is set to 60.

8.4.16 showMessage

[all]

```
Connection.setProperty(“showMessage”, “true”);
```

By default, all debugging and error messages that occur during the running of a JSDT application are suppressed. Setting the *showMessage* property to *true* allows you to write these messages to *debugStream*.

8.4.17 showStack

[all]

```
Connection.setProperty(“showStack”, “true”);
```

When a debugging or error message occurs, if *showStack* is set to *true*, then a stack trace of this thread will be written to *stderr*. By default, this property is set to *false*.

8.4.18 socketFactoryClass

[socket]

```
Connection.setProperty(“socketFactoryClass”,
        “com.sun.media.jsdt.socket.SSLSocketFactory”);
```

This allows you to specify an alternate factory class for creating sockets. This class file must implement the com.sun.media.jsdt.impl.JSDTSocketFactory interface, which defines two methods:

```
    Socket
    createSocket(String address, int port)
                throws IOException, UnknownHostException;

    ServerSocket
    createServerSocket(int port) throws IOException;
```

By default, TCP sockets are used in conjunction with the com.sun.media.jsdt.socket.TCPSocketFactory class.

8.4.19 SSLKeyStore

[socket]

```
Connection.setProperty("SSLKeyStore", "keystore");
```

If you are using SSL sockets (see socketFactoryClass), then this property defines the location of your keystore file.

8.4.20 timeoutPeriod

[all]

```
Connection.setProperty("timeoutPeriod", "3000");
```

When a message is sent from a JSDT proxy to a JSDT server, then if a reply is not received within *timeoutPeriod* milliseconds, then a *TimedOutException* will occur. This property allows you to adjust that timeout period. The default value is 15000 milliseconds (15 seconds).