Integration Services - 1

- Multiple tools available for integration

- Foundation framework
- Connection Server
- Message Objects
- Utilities
- ApplFramework

Connection Server Overview

- Framework for managing TCP/IP connections.
- Pure 100% Java implementation.
- Used for providing communication integration between processes or systems using sockets.
- Used for integrating Legacy/CORBA based applications to Java based applications.
- Provides and exposes an interface for pluggable objects to deal with formatting and processing of messages.
- Integrated to run within the Foundation Framework.
- Connection Server can be configured in every process that requires raw TCP/IP communication with another process or a system.

Connection Server Features

- Protocols supported – TCP and UDP.
- Provides control over socket options (eg socket window sizes).
- Allows configuration for multiple named remote connections, each over a different thread.
- Each connection can be configured as either a Client or a Server connection. (Client connections make remote connections and Server connections allow remote applications to connect)
- Provides automatic failover capability to ‘n’ remote connections.
- Allows grouping of multiple active connections (where only one of them is considered in active state). (Generally used when receiving the same data simultaneously over more then one connection at the same time).
- User provides implementation class(pluggable module) for dealing with messages for each configured connection in the process.
- Allows user to hook in a pluggable object to perform session establishment before forwarding the input/output streams to the implementation class.
- Administrative requests available to start,stop and check the state of the connections.
- Connections can be configured to be automatically started when the process comes up or can be started manually via the administrative requests.
- Provides logging capability (when you log a message using the connection server methods it will automatically add the connection name of the connection logging the alarm).

Connection Server Architechture

- See provided attachments for class diagrams and sequence diagrams for internal workings of the Connection server framework.
- Internal workings are beyond the scope of this course.
- Documentation in rose is in vobs/dte/models/Detail Design/SBT Design Model
- Code and documentation is in package com.cboe.connectionServer
- Code is in /vobs/dte/server/connectionServer.
- Home for connection server is *ConnectionServiceHomeImpl* . You can use this as a starting point to look thru the connectionServer code.

ConnectionServer Framework exposed Components (1)

- ### *Service* Interface (1)

For every configured connection the user must supply an implementation class ***(your pluggable object)*** that deals with the messages received or sent over the connection. This class must implement the ***Service*** interface.

```
public interface Service {
        public void   setName(String aName);
        public void   initialize(String configPrefix,
                                    LogService logService) ;
        public Service getInstance();
        public String getName();
        public void   goSlave();
        public void   goMaster(boolean failingOver);
        public void   start();
        public void   stop();
        public void   serve(InputStream in, OutputStream out,
                             String connectionName)
                             throws IOException;
        public String toString(String indent);
}
```

NOTE: See provided attachments or CODE for more documentation.

- ### *Service* Interface (2)

For every configured connection the connection server will call the methods on the ***Service*** interface in the following order.

When *ConnectionServiceHome* is called with *initialize()* by the *FoundationFramnework*.

1) The connection server framework will first ***instantiate*** the implementation class implementing the *Service* interface.
2) Then ***setName()*** is called with the configured name.
3) *initilize()* method is called, gives impl chance to load properties.

When *ConnectionServiceHome* is called with *goSlave() or goMaster()* by the *FoundationFramnework.*

4) *goSlave() or goMaster()* methods are called

When *ConnectionServiceHome* starts or stops a connection.

6) *start() or stop()* methods are called.

When *ConnectionServiceHome* **establishes** a connection..

7) *getInstance() method is called* and then *serve() method* is called on the object returned by *getInstance()*. (This is the processing while loop).

On Admin request to display connection information.

8) *toString() method* is called.

ConnectionServer Framework exposed Components (2)

- *DefaultService* Abstract class (1)

  For every configured connection the user can either directly implement the *Service* interface or they **may extend** this abstract class.

  The methods in this class provide some base implementation for the *Service* interface.

```
abstract class DefaultService implements Service {
public String getName();
public void    setName(String aName);
public void    initialize(String aConfigPrefix,
                          LogService aLogService)
public void    logDebug(String componentName, String text)
public void    logError(String componentName,String text)
public void    logException(String componentName,
                            String text,  Throwable t)
public void     logInformation(String componentName,
                               String text)
public void     logWarning(String componentName,
                           String text)
}
```

NOTE: See provided attachments or CODE for more documentation.

- *DefaultService* Abstract class (2)

  This class implements the Service interface and provides implementation for commonly used methods and provides helper methods for logging alarms, information and debug messages.

  NOTE:

  The *initialize()* method sets up the base property name for easy lookups.

  The sub-class can override & call this method. If called then the internal variable *configPrefix* is setup with the full home name + "." the connection name + "."

  Later on when a property is needed the sub class can just the property name and call the *configService* to get the property value.

ConnectionServer Framework exposed Components (3)

SessionManager Abstract class (1)

This abstract class is used by the connection server to perform session establishment before forwarding the input/output streams to the underlying service.

Associated with each named connection there **MAY Optionally** be a *SessionManager* registered in the properties/XML configuration file.

The session manager impl then does what ever is needed to establish the session appropriately.

Usage on how the implementation class should code to this abstract class.

1. The implementation must extend this class.
2. The implementation must provide a constructor that takes no argument's. The implementation is then invoked on the <code>setName(String connectionName)</code> method.
3. The implementation is first invoked with the initialize() method. At this point the implementation can initialize itself [like reading configuration information].
4. Every time the connection is established/re-established the implementation is then invoked with the establishSession() method. [At this point the implementation does what ever is needed to establish the session with the remote side].

If unable to establish the session then the implmentation should throw exceptions.

SessionManager Abstract Class (2)

```
public abstract class SessionManager {
   public void establishSession(DatagramSocket aSocket)
                                 throws IOException
   public abstract void establishSession(Socket aSocket)
                                 throws IOException;
   public void initialize(String aConfigPrefix,
                          LogService aLogService)
   public void logDebug(String componentName,
                        String text)
   public void logError(String componentName,
                        String text)
   public void logInformation(String componentName,
                              String text)
   public void logWarning(String componentName,
                          String text)
   public void setName(String aConnectionName)
   public String toString(String indent)
}
```

ConnectionServer Framework exposed Components (4)

ConnectionServiceHome

The main BOHome for the connection server framework that sets up the whole framework by reading all the configuration information.

```
public interface ConnectionServiceHome {
    public ConnectionService find();
}
```

ConnectionService

The BObject associated with the above home.

```
public interface ConnectionService {
    public Service findService(String aServiceName);
    public String   startService(String aServiceName);
    public String   stopService(String aServiceName);
}
```

- User can find the ConnectionService Home and the ConnectionService Objects programmatically inside their code.
- From your application code you can
  - Find your named connections
  - start the named connection
  - Start the named connection

ConnectionServer Framework exposed Components (5)

- Code example – How to find ConnectionServiceHome

  In some application home code

  HomeFactory factory = HomeFactory.getInstance();

  connectionServiceHome =
  (ConnectionServiceHome)factory.findHome(ConnectionServiceHome.HOME_NAME);

  connectionService = connectionServiceHome.find();

  Given the the ConnectionService Object you can start and stop your connections programmatically.

Message Objects (1)

- Used for encoding and decoding protocols.

<u>*Base Classes & Interface*</u>

- *Message* Interface
- *MessageInputStream* Abstract
- *MessageOutputStream* Abstract

<u>*CBOE Protocol Messages*</u>

- NapiMessage
- NapiInputStream
- NapiOutputStream

- Base classes are in package
  - com/cboe/message
  - Directory: /vobs/dte/server/message

- Napi implementation is in package
  - com/cboe/napi
  - Directory: /vobs/dte/server/message

Message Objects (2)

Message Interface (1)

- This is the base interface used for encoding and decoding of a specific protocol.
- Associated with this interface is the *MessageInputStream* and *MessageOutputStream* that are used to read and write messages of type *Message*.
- public interface *Message* {

```
  public byte[] getBytes();
  public int      getLength();
  public int      getMaximumLength();
  public void    setMaximumLength(int maxLen);
}
```

Message Interface (2)

- When reading and writing messages you can provide an implementation of this interface that will be responsible for holding and encoding and decoding the messages you send or receive from the socket input / output stream.
- The *MessageInputStream* and *MessageOutputStream* classes use objects of type *Message* to send and receive data from the underlying input/output streams. The stream classes use the *Message* objects to provide encoding and decoding of Messages.
- Currently the 2 concrete implementation's available are
  - NapiMessage
  - ApplMessage

Message Objects (3)

MessageInputStream Abstract

- public abstract class *MessageInputStream* {
  - public MessageInputStream()
  - public MessageInputStream(String commPathName)
  - public void close() throws IOExceptio
  - public Message readMessage() throws IOException
  - protected void updateCommpathError(Exception e)
  - protected void updateCommpathMessageReceived(int msgLength)
  - **protected abstract Message localReadMessage() throws IOException;**
- }
- Responsible for decoding protocol messages of type *Message.*
- Implementation must extend this class and provide implementation for method *localReadMessage.*
- Implementation must also provide a constructor that will take in **at least** a *java.io.InputStream* as an argument

MessageOutputStream Abstract

- public abstract class MessageOutputStream {
  - public MessageOutputStream()
  - public MessageOutputStream(String commPathName)
  - public void close() throws IOException
  - protected void updateCommpathError(Exception e)
  - protected void updateCommpathMessageSent(int msgLength)
  - public synchronized void writeMessage(Message aMessage) throws IOException
  - **protected abstract void localWriteMessage(Message aMessage) throws IOException;**
  - }
- Responsible for decoding protocol messages of type *Message.*
- Implementation must extend this class and provide implementation for method *localWriteMessage.*
- Implementation must also provide a constructor that will take in **at least** a *java.io.OutputStream* as an argument

Message Objects (4)

- *NapiMessage*
  - implements *Message*
  - This Object is a concrete implementation that deals with the encoding and decoding of the Napi protocol.
- *NapiProtocol*

  Message Layout is as follows

  Msg Length - 2 bytes short

  Data - (1 – n bytes)
- *NapiInputStream/NapiOutputStream*
  - *Implement the MessageInputStream and MessageOutputStream interfaces respectively.*
  - *Resonsible for reading and writing NapiMessages.*
- *ApplMessage – Another example.*
  - *Another concrete implementation of CBOE's appl-appl protocol.*
  - *Appl-Appl protocol*

    Version(0,1) MsgType(1,1) Origin(2,8) Destination(10,8) Key(18,4) Data(22,n)

```
public ApplMessage implements Message {
        public ApplMessage constructReply(…)
        public void copyToApplHeader()
        public void copyToData()
        public static ApplMessage createApplMessage()
        public static ApplMessage createApplMessage()
        public static ApplMessage createApplMessage()
        public byte[] getApplicationHeader()
        public int getApplicationHeaderLength()
        public byte[] getBytes()
        public int getCommand()
        public byte[] getData()
        public int getDataLength()
        public String getDestination()
        ……..
}
```

ConnectionServer Configuring Connections (1)

- This section explains how to setup & configure connections managed by the ConnectionServer framework.
- To configure connections you need to setup the DTD and XML configuration for the process where the connections will reside in.
- The whole Connection Server framework is setup as a home in a process.
- Hence for each time you configure a connection server home for a particular process, you need to define the connections that will be managed by that home in that process.

- ConnectionService.dtd (1)
  - Is a Base DTD needed for all connection server configurations.
  - Defines the properties needed to configure a connection.
  - User needs to create their own dtd that augments this dtd to define their connection names.
  - User can add their own application specific properties for each connection.
  - Directory : /vobs/dte/server/release/dtd

ConnectionServer Configuring Connections (2)

- ConnectionService.dtd (2)
- Properties for ConnectionServiceHome
  - dropConnectionsWhenSlave
    - If ‘true’ we will drop all all connections when goSlave() is called by foundation framework.
  - serviceNames
    - The names of all your configured connections.

- ConnectionService.dtd (3)
- Properties for each *serviceName*
  - protocol
    - tcp or udp
  - serviceType
    - client or server
  - serviceImpl
    - Full patch of the class implementing the *Service* interface
  - enabled
    - If ‘true’ then the connection will automatically be started when the process comes up. (used in conjunction with *dropConnectionsWhenSlave*)
  - bindHost
    - For server socket the tcp listen host.
  - bindPort
    - For server socket the tcp listen host.

ConnectionServer Configuring Connections (3)

- ConnectionService.dtd (4)
  - hostNames
    - For client connections, a comma separated list of remote host names.
  - portNumbers
    - For client connections, a comma separated list of remote port nbrs.
  - retryTimeout
    - Sleep time in milliseconds between recovery to the next host/port combination.
  - retryCount
    - Nbr of times to retry the same connection before trying the next host/port combination.

- ConnectionService.dtd (5)
  - maxInputQueueDepth
    - For udp connections only. Max nbr of queued msgs after which msgs will start being discarded.
  - socketSendBufferSize
    - Send socket window size. Higher the better.
  - socketRecvBufferSize
    - Recv socket window size. Higher the better.
  - inputStreamBufferSize
    - Read buffer length (used to buffer socket streams).
  - outputStreamBufferSize
    - Write buffer length (used to buffer socket streams).

ConnectionServer Configuring Connections (4)

- ConnectionService.dtd (6)
  - sessionManager
    - Optional. Name of the class that will deal with the session establishment before input/output streams are passed to the *serviceImpl.*

- To reiterate configuration
  - Create your own dtd
  - Specify list of connections in the dtd.
  - If your implementation has admin requests or properties then add them to this dtd.
  - Create your xml with the appropriate values.
  - Next section we will show a concrete example.

ConnectionServer Exposed Admin requests (2)

showContext

- Display's information about one or more connections
- Usage:
  - ***showContext [serviceName optional]***
  - Where *serviceName* is the the name of your connection
  - Partial names are allowed. If a partial name is given then information about all services matching the partial name will be shown.
- Example:

```
ar TradeReportAdapter showContext TradeReport
```

startService

- Tells the connection server to start the specified service.
- Usage:
  - ***startService serviceName***
  - Where *serviceName* is the the full name of your connection
- Example:

```
ar TradeReportAdapter startService
        TradeReportServiceConnection
```

ConnectionServer Exposed Admin requests (2)

stopService

- Tells the connection server to stop the specified service, by *interrupting* the thread that is in the while loop reading from the inputStream in the *serve*() method.
- Usage:
  - ***stopService serviceName***
  - Where *serviceName* is the the full name of your connection
  - Connection server will wait for 5 seconds for the application to return from the *serve*() method**, after that the connection will be closed forcefully.**
- Example:

```
ar TradeReportAdapter stopService
    TradeReportServiceConnection
```

startAllServices

- Tells the connection server to start one or more services.
- Usage:
  - ***startAllServices [serviceName optional]***
  - Where *serviceName* is the the name of your connection
  - Partial names are allowed. If a partial name is given then all services matching the partial name will be started.
  - If no names are given then all connections are started.
- Example:
  - ar TradeReportAdapter startAllServices
  - ar TradeReportAdapter startAllServices TradeReport

ConnectionServer Exposed Admin requests (3)

stopAllServices

- Tells the connection server to stop one or more services.
- Usage:
  - ***stopAllServices [serviceName optional]***
  - Where *serviceName* is the the name of your connection
  - Partial names are allowed. If a partial name is given then all services matching the partial name will be stopped.
  - If no names are given then all connections are stopped.
- Example:
  - ar TradeReportAdapter stopAllServices
  - ar TradeReportAdapter stopAllServices TradeReport

- User can add their own admin requests
- Admin requests are to be specified in the dtd.
- User has to use CommandCallbackService directly to register their own admin requests.
- User will need to specify for which connection the admin request is applicable at the time of registration.
- More in the examples.

TradeReportAdapter Development – An Example

- TradeReportAdapter
  - Receive’s trade reports from an event channel and sends it to the remote system for TradeMatch.
  - Connection will register an admin request.
  - Connection will define additional properties.
  - Receiving end configured in the process, that simulates the TradeMatch system (Just logs all trade reports to a file).
- What are the steps
  - Code a java class that implements the *Service* interface and extends the *DefaultService*.
    - Example: TradeReportServiceImpl
  - Define connections in a separate dtd file.
    - Example: TradeReportConnectionService.dtd
  - Specify values in the xml for the new connections.
    - Example: TradeReportConnectionService.xml
  - Define the ConnectionService Home & Other needed homes in the *Foundation Framework* process you will run this code in.
    - Example: TradeReportAdapter.xml

TradeReportAdapter Development Implementing *Service* Interface (1) TradeReportServiceImpl.java

```
public void initialize (
    String  aConfigPrefix,
    LogService aLogService)
    throws   NoSuchPropertyException {
    // CALL SUPER CLASS TO SETUP THE CONFIG
    // PREFIX
    super.initialize(aConfigPrefix,aLogService);

    // regsiter admin request to send a manual text message.
    CommandCallbackService ccs =
    FoundationFramework.getInstance().getCommandCallb
    ackService();
    String commandName = configPrefix + sendText;
    try {
        ccs.registerCommandCallback(this, commandName,
    }
    catch (Exception e) {
        logError (myClassName,”Error registering …”);
        throw new IllegalArgumentException (e);
    }
```

```
// get Property, trade report channel name
ConfigurationService configService =
FoundationFramework.getInstance().getConfigService();
String propertyName = “”;
try {
    propertyName = configPrefix + “tradeReportChannel”;
    channelName  = config.getProperty(propertyName);
    ……… get other properties.
}
catch (Exception e) {
    logException(myClassName,
                    “Error getting property: ” + propertyName);
    throw new IllegalArgumentException (“” + e);
}

// Create the trade report consumer that will connect to the event
// channel. We will use this later to connect and disconnect from
// the trade report event channel.
myTradeReportConsumer = new TradeReportConsumer(this);
```

TradeReportAdapter Development Implementing *Service* Interface (2) TradeReportServiceImpl.java

```
public void start() {
    // Is called when the connection is
    // started (Connection is not
    // necessarily connected at this
    // point in time)
    // Nothing special I need to do here in this
    // case.
}
public void stop() {
    // Is called when the connection is
    // stopped (Connection is not
    // necessarily dis-connected at this
    // point in time)
    // Nothing special I need to do here in this
    // case.
}
```

```
public void goSlave() {
// No specific connection server requirements
// Upto the implementor to do what is needed
// for the application when it goes slave.
// If you don't want the slave to receive data.
// then generally you would disconnect from
// the channel, disconnect from the POA etc..
// In our example here we will disconnect from
// the trade report channel, if we running in
// slave mode. Generally this step would be done
// by some consumer home as explained in the
// FoundationFramework course.

    EventService eventService =
    FoundationFramework.getInstance().getEventService();

    eventService.disconnectNotificationConsumer(myTrade
    ReportEventConsumer);
    eventConsumerRegistered = false;
}
```

TradeReportAdapter Development Implementing *Service* Interface (3) TradeReportServiceImpl.java

```
public void goMaster() {
        // No connection server requirements here
        // Upto the implementor to do what is needed
        // for the application when it goes master.
        // Connect to channels, export your objects
        // In our example here we will connect to the
        // consumer. Generally this step would be done
        // by some consumer home as explained in the
        // FoundationFramework course.
        if (eventConsumerAlreadyRegistered)
            return;
        String repositoryId =
        TradeReportEventConsumerHelper.id();

        String  logString = eventChannelName + "/“
                                                repositoryId;
        EventService eventService =
        FoundationFramework.getInstance().getEventService();
try {
        logInformation(myClassName, "Registering consumer”)

        eventService.connectTypedNotifyChannelConsumer(ch
        annelName, repositoryId,
        myTradeReportEventConsumer);

        eventConsumerRegistered = true;
        logInformation(myClassName, “Registered consumer”);
}
catch (Exception e)
{
        logException(myClassName,"Unable to register event
        consumer : " + logString, e);
        eventService.disconnectNotificationConsumer(myTrade
        ReportEventConsumer);
}
```

TradeReportAdapter Development Implementing *Service* Interface (4) TradeReportServiceImpl.java

```
public void serve (java.io.InputStream in ,
                  java.io.OutputStream out,
                  String connectionName)
        throws IOException {
    // This is the main processing loop.
    // This thread primary responsibility if to read messages
    // of the input stream and process them.
    // IN OUR EXAMPLE WE ARE NOT EXPECTING TO
    // RECEIVE ANYTHING.
    // BUT STILL we cannot return back from the call otherwise
    // connection server will close the underlying socket and try
    // to recover the connection.

    // Create infrastructure 'commpath' for message rate
    // collections. Do this if you want your message statistics to be
    // monitored
    String commpathName = myName + "-" + connectionName;

    // Create napi streams.
    inStream  = new NapiInputStream(in,commpathName);
    outStream = new NapiOutputStream(out,commpathName);

    // Process messages.
    try  {
        // THE MAIN LOOP.
        // IF YOU ARE NOT DISPATCHING TO THREADS
        // THEN YOU NEED CREATE ONLY ONE NapiMessage
        // OBJECT OTHERWISE CREATE ONE EVERY TIME.
        while (!Thread.currentThread().isInterrupted()) {
            // Create napi message to read the messages into.
            NapiMessage napiMessage =
                new NapiMessage(tipsMessageMaxLen);
                inStream.readInto(napiMessage);

                // Process and do whatever you want to
                handleMessage(napiMessage);
        }
    }
    finally {
        inStream.close();
        outStream.close();
    }
```

TradeReportAdapter Development Implementing *Service* Interface (5) TradeReportServiceImpl.java

```
public Service getInstance() {
        // If you are a server generally you would
        // return a new class every time.
        // The connection server will call the serve()
        // method on the object returned by this
        // method.
        // Hence, be careful when coding a server.
        // In our case we will assume we are a
          client.
            return this;
        }
```

```
public String toString (String indent) {
        // Return a string representation of
        //  information you want to see
        // This information is returned when
        // ar TradeReportAdapter showContext
        // is executed on the command line.

        String returnValue = “”;
        returnValue += “\n” + indent;
        returnValue += “Channel = “ + channelName;
        returnValue += “\n” + indent;
        returnValue += “ConsumerRegistered = “ +
                        eventConsumerRegistered;
```

TradeReportAdapter Development TradeReportServiceImpl.java

```
public void sendMessage(Message
    aMessage) throws IOException {
    // Just a publicly available method for other
    // objects to send messages.
    // Send the message out over the outputStream that
    // was saved in the serve() method.

    outStream.writeMessage(aMessage);
}
```

```
public String sendText(String aText) {
    // The admin request called when some body uses
    // the command line to manually send requests to this
    // connection.

    // Here we will create a message and send it out over
    // the wire to the other system.
    String text = “Some test sent to other side”;
    NapiMessage textMessage =
                    new NapiMessage(text);
    sendMessage(textMessage);
}
```

TradeReportAdapter Development MyTradeReportEventConsumer.java

```
class MyTradeReportEventConsumer
    extends
    POA_TradeReportEventConsumer
// This class should generally be coded as a
// home and is used just as an example to
// complete this course. Generally you want
// some home to connect your consumers
// etc…
MyTradeReportEventConsumer(TradeRepo
    rtServiceImpl serviceImpl) {
    // This constructor just holds onto the Service impl
    mySeriviceImpl = serviceImpl;
}
```

```
acceptTradeReport(TradeReportStruct s) {
    // This method receives a trade report , formats it according to
    // the format the remote system is expecting it and sends it to the
    //  connection registered with this object.

    // FORMAT THE DATA YOUR OWN WAY.
    String s = “Received Trade Report “;
    NapiMessage message = new NapiMessage(s);

    // Now we send it to the connection server
    try {
            myServiceImpl.sendMessage(message);
    }
    catch (IOException e) {
            // Upto you to as to what you want to do if there
            // is no connection.
            // There are tools available that you can use to
            //  queue messages, as shown later.
    }
```

TradeReportAdapter Development TradeReportConnectionService.dtd

Defines connections and their properties

```
<!-- define the entities required by the ThisConnectionService -->
<!-- Here we list the names of all the connections that will be →
<!-- configured for this feed -->
<!ENTITY % applicationDefinedServiceConnectionServices
                    "(TradeReportServiceConnection)" >

<!-- include the ConnectionService -->
<!ENTITY % CS_include SYSTEM "ConnectionService.dtd" >
%CS_include;

<!ELEMENT TradeReportServiceConnection
                    (TradeReportServiceConnectionProperties,
                    TradeReportServiceConnectionCommands?)>
<!ATTLIST TradeReportServiceConnection
        type %typeConstraints; #FIXED "ManagedResource"
        objectName CDATA #IMPLIED
>
<!ELEMENT
        TradeReportServiceConnectionProperties(tradeReportChanne
        l) >
<!ATTLIST TradeReportServiceConnectionProperties
        type %typeConstraints;  #FIXED "ManagedProperties“
>
```

```
<!ELEMENT tradeReportChannel (#PCDATA)>
<!ATTLIST   tradeReportChannel
   type            %typeConstraints;    #FIXED "ManagedProperty"
   mode            %modeConstraints;    #FIXED "readWrite"
   propertyType    %propertyTypes;      #FIXED "string"
   description     CDATA                #FIXED “ channel name”
>
<!-- configure your admin request -->
<!ELEMENT TradeReportServiceConnectionCommands (sendText)>
<!ATTLIST   TradeReportServiceConnectionCommands
      type       %typeConstraints;       #FIXED    'ManagedCommands'
>
<!ELEMENT sendText (text) >
<!ATTLIST   sendText
      type          %typeConstraints;   #FIXED     'ManagedCommand'
      description CDATA                 #FIXED     ‘Send a text message'
>
<!ELEMENT text ANY >
<!ATTLIST text
      type             %typeConstraints;  #FIXED 'ManagedParameter'
      parameterType    %propertyTypes;    #FIXED 'string'
     description       CDATA              #FIXED ‘some text'
>
```

TradeReportAdapter Development TradeReportConnectionService.xml

Define the real values in the xml

```
<?xml version="1.0" encoding="UTF-8"?>
<!-- edited with XML Spy v3.0.7 NT (http://www.xmlspy.com) by
        (Chicago Board Options Exchange) -->
<!DOCTYPE GlobalConnectionService SYSTEM
                    "../dtd/TradeReportConnectionService.dtd">
<GlobalConnectionService>
   <ConnectionServiceHomeImpl
                    name="ConnectionServiceHomeImpl">
        <ConnectionServiceHomeImplProperties>
             <dropConnectionsWhenSlave booleanValue="true"/>
             <serviceNames>TradeReportConnection</serviceNames>
        </ConnectionServiceHomeImplProperties>

        <Service name="TradeReportServiceConnection“>
             <ServiceProperties>
                 <protocol protocolType="tcp"/>
                 <serviceType serverType="client"/>
                 <serviceImpl> com…TradeReportServiceImpl
                                              </serviceImpl>
                   <enabled booleanValue="false"/>
                   <hostNames>localhost,localhost</hostNames>

                   <portNumbers>40000,40000</portNumbers>
                   <retryTimeout>7000</retryTimeout>
                   <retryCount>1</retryCount>
                   <maxInputQueueDepth>-1</maxInputQueueDepth>
                   <socketSendBufferSize>63556</socketSendBufferSize>
                   <socketRecvBufferSize>63556</socketRecvBufferSize>
                   <inputStreamBufferSize>32768</inputStreamBufferSize>
                   <outputStreamBufferSize>4096</outputStreamBufferSize>
             </ServiceProperties>
        </Service>

        <TradeReportConnection>
             <TradeReportConnectionProperties>
                  <tradeReportChannel>TradeReport</sessionFilter>
             </TradeReportConnectionProperties>
        </TradeReportConnection>
   </ConnectionServiceHomeImpl>
</GlobalConnectionService>
```

TradeReportAdapter Development
TradeReportAdapter.dtd
TradeReportAdapter.xml

TradeReportAdapter.dtd

1) DTD for the whole process
2) Looks like any other process as explained in the foundation framework course.
3) Important elements for connection server highlighted in bold.
4) See provided attachment

TradeReportAdapter.xml

1) XML for the whole process
2) Looks like any other process as explained in the foundation framework course.
3) Important elements for connection server highlighted in bold.
4) See provided attachment

Utilities (1)

NapiMessage *implements Message*

- Package *com.cboe.napi*
- Message encoder/decoder
- Message layout is
  - 2 bytes binary message length
  - ‘n’ bytes of data.

```
public class NapiMessage implements Message {

    public NapiMessage()
    public NapiMessage(byte[] bytes)
    public NapiMessage(byte[] b, int offset, int length)
    public NapiMessage(int maxLen)
    public NapiMessage(String source)
    public byte[] getBytes()
    public int getLength()
    public int getMaximumLength()
    public void setMaximumLength(int newValue)
    public static void readInto  (NapiMessage message,
                                  DataInputStream dataStream)
                                  throws IOException

}
```

Utilities (2)

NapiInputStream

- Package *com.cboe.napi*
- Used for reading NapiMessages

```
public class NapiInputStream extends
                MessageInputStream {

    public NapiInputStream(java.io.InputStream in)
    public NapiInputStream(InputStream in,  String aName)
    public void close() throws IOException
    public synchronized void readInto(NapiMessage msg)
                                    throws IOException
}
```

NapiOutputStream

- Package *com.cboe.napi*
- Used for writing NapiMessages

```
public class NapiOutputStream extends
                MessageOutputStream {
    public NapiOutputStream(OutputStream out)
    public NapiOutputStream(OutputStream out, String aName)
    public void close() throws IOException
    public void flush() throws IOException
}
```

Utilities (3)

Persistent Sequencer - Home

- Package com.cboe/externalIntegrationServices.utils
- Used for finding and creating a sequencer
- Supports file based or database based persistence.
- Specific home needs to configured in your process.

```
public interface SequencerHome {
    public Sequencer create(String aName) throws IOException;
    public Sequencer find(String aName) throws IOException;
    public void resync(Sequencer aSequencer) throws IOException;
    public void resyncAll() throws IOException;
}
```

Persistent Sequencer – Sequencer Interface

- Package com.cboe/externalIntegrationServices.utils
- Used for persisting messages
- Persists only one sent and one received message
- Keeps track of sequence nbrs.

```
public interface Sequencer {
    public int getExpectedSequenceNbr();
    public byte[] getLastSentMessage();
    public int getLastSentMessageLength();
    public int getNextSendSequenceNbr();
    public int getReceiveSequenceNbr();
    public int getSendSequenceNbr();
    public String getSequencerName();
    public void processReceivedMessage(int sequenceNbr) throws
    IOException;
    public void processReceivedMessage(int sequenceNbr,boolean
    synchronizeRemote) throws IOException;
    public void processSentMessage(byte[] message, int offset, int
    length, int sequenceNbr) throws IOException;
    public void processSentMessage(byte[] message, int offset, int
    length, int sequenceNbr, boolean synchronizeRemote) throws
    IOException;
    public void receiveMessage(int sequenceNbr) throws
    SequenceNbrTooHighException,SequenceNbrTooLowExceptio
    n; ……..
```

Utilities (4)

Formatter

- Package com.cboe.externalIntegrationServices.utils
- Utility used for converting data in byte[] to/from native data types.

public class Formatter {

public static boolean isStringNumeric(String aString)

public static final int readByteAsInt(byte[] data)

public static final int readByteAsInt(byte[] data, int offset)

public static final int readInt(byte[] data)

public static final int readInt(byte[] data, int offset)

public static final long readLong(byte[] data)

public static final long readLong(byte[] data, int offset)

public static final int readNbytesAsInt(byte[] data,int offset, int length)

public static final long readNbytesAsLong(byte[] data,int offset, int length)

Formatter cont'd

public static final int readShortAsInt(byte[] data)

public static final int readShortAsInt(byte[] data, int offset)

public static int readStringAsInt(byte[] data, int offset, int length)

public static long readStringAsLong(byte[] data, int offset, int length)

public static final void writeInt(int v, byte[] data, int offset)

public static void writeIntAsString(int value, byte[] data, int offset, int length)

public static final void writeLong(long v, byte[] data, int offset)

public static void writeLongAsString(long value, byte[] data, int offset, int length)

public static final void writeNbytesInt(int v, byte[] data, int offset, int length)

public static final void writeNbytesLong(long v, byte[] data, int offset, int length)

public static final void writeShort(int v, byte[] data, int offset)

public static void writeSpaceFilledString(String value, byte[] data, int offset, int length)

Utilities (5)

Queue Home

- Package: com.cboe.infrastructureServices.queue
- Used for creating either persistent or transient queues.
- ***Transactional (Works within a Jgrinder Transaction)***

```
public class QueueHome extends BOHome
                                 implements Timer {
// THESE ARE THE ADMINSTRATIVE COMMANDS THAT
// CAN BE GIVEN ON THE COMMAND LINE
      public String adminClearQueue(String queueName)
      public String adminCreateQueue(String queueName,
                                   String persistentOrTransient)
      public String adminDequeueElement(String queueName)
      public String adminDumpQueue(String queueName,
                                   String briefOrVerbose)
      public String adminEnqueueElement(String queueName,
                                      String stringToEnqueue)
      public String adminStartMonitoringQueues(String queueName)
      public String adminStopMonitoringQueues(String queueName)
```

Queue Home cont’d

```
// THESE ARE THE METHODS USED FOR FINDING
// AND CREATING QUEUES.
public synchronized Queue create(String queueName,
                                 boolean isPersistent)
                                   throws QueueException
public Queue createTransient()
public synchronized Queue find(String queueName)
                                   throws QueueException
public synchronized static QueueHome getInstance()
```

Utilities (6)

Queue Interface

- Package: com.cboe.infrastructureService.queue
- Used to enqueing/dequeing elements of the queue

```
public interface Queue {
    void clear() throws QueueException;
    Object dequeue() throws QueueException;
    Object dequeue(int waitMillis) throws QueueException;
    Object[] dequeueMultiple(int nbrToDequeue)
                                throws QueueException;
    Object[] dequeueMultiple(int waitMillis,int nbrToDequeue)
                                throws QueueException;
    void enqueue(Object dataObject) throws QueueException;
    void enqueue(Object dataObject, int waitMillis)
                                throws QueueException;
    int getDefaultTimeout();
    int getMaxQueueDepth();
    String getQueueName();
    boolean isEmpty() throws QueueException;
```

Queue cont'd

```
    boolean isFull() throws QueueException;
    Object peek() throws QueueException;
    Object peek(int timeoutMillis) throws QueueException;
    Object[] peekMultiple(int nbrToPeek) throws QueueException;
    public Object[] peekMultiple(int newWaitTime, int
    nbrToDequeue) throws QueueException;
    void setDefaultTimeout(int timeoutMillis);
    void setMaxQueueDepth(int maxQueueDepth) throws
    QueueException;
    int size();

}
```

Appl Framework Overview (1)

- Cboe proprietary protocol.
- Allows for multiplexing and demultiplexing of logical connection’s over the same Socket connection
- Composed of client side and server side.
- Client’s login (with their own unique name) to a named service
- Server exposes a named service.
- Protocol provides mechanism for GMD & sequence accounting and failover capabilities.

- Appl Framework
  - Another framework like connection server.
  - A very concrete implementation.
  - Uses connection server as its underlying transport.
  - Just like the connection server this framework its has its own home ApplFrameworkHome that sets up the whole framework.
  - Exposes interfaces and classes just like the connection server.

Appl Framework Overview (2)

- Documentation in rose is in vobs/dte/models/Detail Design/SBT Design Model
- Code and documentation is in package com.cboe.externalIntegrationServices.applappl
- Code is in /vobs/dte/server/applappl
- Home for connection server is *ApplFrameworkHomeImpl* . You can use this as a starting point to look thru the ApplFramework code.

Appl Framework Protocol Overview (1)

- Version 1 Layout
  - VERSION1 has the following message layout
  - Version(1 byte) MsgType(1 byte) Origin(8 bytes) Destination(8 bytes) Key(4 bytes) Data(22,n)
- Version 2 Layout
  - VERSION2+ has the following message layout
  - Version(1 byte) MsgType(1 byte) Origin(8 bytes) Destination(8 bytes) Key(4 bytes) HdrLen(1 byte) ApplHdr(n bytes) Data(m bytes).

Appl Framework Protocol Overview (2)

- Message Types
  - Following messages are supported (in the **MsgType** field from previous slide)
    - CONNECT_PRIMARY
    - CONNECT_SECONDARY
    - CONNECT_ACCEPT
    - CONNECT_REJECT
    - DISCONNECT_PRIMARY
    - DISCONNECT_SECONDARY
    - DISCONNECT_ACCEPT
    - DATA_REJECT
    - DATA_WITH_CONFIRM
    - CONFIRM_RESPONSE
    - HEARTBEAT_REQUEST
    - HEARTBEAT_RESPONSE
    - DISCONNECT_REJECT
- Flow
  - See provided attachments to see how the how the client and server components interact.

Appl Framework Exposed Components Overview (1)

- *ApplMessage* – Message encoder/decoder.
- *ApplService interface* – User implements this interface to provide either client side or server side implementation to deal with logins, sequence accounting, multiplexing of logical client logins etc…
- *ApplFrameworkImpl* - is reponsible for decoding the ApplMessage, finding the *ApplService* Object responsible for handling that message(based on login names) and calling appropriate methods on the *ApplService* Interface.
- *ApplDefaultClient* – Default implementation of the *ApplService* interface. Deals with GMD, sequence accounting, logins, hearbeats, login timeouts, confirm responses.
- *ApplDefaultService* – Default implementation for server side

- Client side can extend ApplDefaultClient and overwride methods if needed.
- Client and server side must deal with internal formatting and processing of message.
- Server side can extend ApplDefaultService and overwride methods if needed.

Appl Framework Exposed Components Overview (2)

ApplMessage {

```
public ApplMessage
        constructReply(intcommand,byte[]data,intoffset,intlength)
public void copyToApplHeader(byte[]newApplHeader,intoffset,intlength)
public void copyToData(byte[]newData,intoffset,intlength)
public static ApplMessage
        createApplMessage(byte[]message,intmessageLength)
public static ApplMessage
        createApplMessage(intversion,intcommand,StringanOrigin,StringaDest
        ination,byte[]aKey)
public static ApplMessage createApplMessage(MessageinputMessage)
public static ApplMessage
        createApplMessage(StringpropertiesFile)throwsException
public byte[] getApplicationHeader()
public int getApplicationHeaderLength()
public byte[] getBytes()
public int getCommand()
public static String getCommandName(intcommand)
public byte[] getData()
public int getDataLength()
public String getDestination()
public int getHeaderLength()
public byte[] getKey()
public int getLength()
public int getMaximumLength()
public String getOrigin()
public int getVersion()
public static booleanisValidLength(intmessageLength,byte[]messageData)
```

ApplService interface

```
public void goMaster(boolean failingOver);
public void goSlave();
public void handleConfirmResponse(ApplMessage theMessage,
        ApplConnection connection ) throws IOException;
public void handleConnectAccept(ApplMessage theMessage,
        ApplConnection connection ) throws IOException;
public void handleConnectReject(ApplMessage theMessage,
        ApplConnection connection ) throws IOException;
public boolean handleData(ApplMessage theMessage,
        ApplConnection connection ) throws IOException;
public void handleDataReject(ApplMessage theMessage,
        ApplConnection connection ) throws IOException;
public void handleDataWithConfirm(ApplMessage theMessage,
        ApplConnection connection ) throws IOException;
public void handleDisconnectAccept(ApplMessage theMessage,
        ApplConnection connection ) throws IOException;
public void handleDisconnectReject(ApplMessage theMessage,
        ApplConnection connection ) throws IOException;
public void handleHeartbeatRequest(ApplMessage theMessage,
        ApplConnection connection ) throws IOException;
public void handleHeartbeatResponse(ApplMessage theMessage,
        ApplConnection connection ) throws IOException;
public void handlePrimaryConnect(ApplMessage theMessage,
        ApplConnection connection) throws IOException;
```

Appl Framework Exposed Components Overview (3)

ApplService Interface cont'd

```
public void handlePrimaryDisconnect(ApplMessage theMessage,
        ApplConnection connection ) throws IOException;
public void handleSecondaryConnect(ApplMessage theMessage,
        ApplConnection connection) throws IOException;
public void handleSecondaryDisconnect(ApplMessage theMessage,
        ApplConnection connection ) throws IOException;
public void handleStart(ApplConnection aConnection);
public void handleStop(ApplConnection aConnection);
public void handleUnknown(ApplMessage theMessage,
        ApplConnection connection) throws IOException;
public void initialize(String configPrefix,LogService aLogService)
        throws NoSuchPropertyException;
public void sendMessage(ApplMessage aMessage) throws
        IOException;
public String setRecvSequenceNbr(int newSequenceNbr,
        ApplConnection aConnection);
public String setSendSequenceNbr(int newSequenceNbr,
        ApplConnection aConnection);
public String toString(String indent);
```